Perforce 2002.1
System Administrator's Guide

April 2002

Table of Contents

Preface

About This Manual

This is the *Perforce 2002.1 System Administrator's Guide*.

It describes the installation, configuration, and operation of a Perforce server from an administrator's perspective. This includes the set of tasks typically performed by a "system administrator" (for instance, installation and configuration of the software, as well as ensuring uptime and data integrity), as well as those performed by a "Perforce administrator", including setting up Perforce users, setting Perforce depot access controls, resetting Perforce user passwords, and so on.

Since Perforce requires no special system permissions, the Perforce administrator typically does not require root-level access. Depending on your site's needs, your Perforce administrator need not be your system administrator.

Both the UNIX and Windows versions of the Perforce server are administered from the command line. If you are unfamiliar with the command line interface to Perforce, you will find the *Perforce Command Reference*, which describes the Perforce command line interface in detail, to be a good companion to this guide.

Although this guide will teach you how to administer a Perforce server, it won't teach you the basics of actually using Perforce. You may also wish to consult the *Perforce User's Guide* for more information on Perforce from a user's perspective.

All our documentation is available from our web site at `http://www.perforce.com`.

Please Give Us Feedback

We are interested in receiving opinions on it from our users. In particular, we'd like to hear from users who have never used Perforce before. Does this guide teach the topic well? Please let us know what you think; we can be reached at `manual@perforce.com`.

Chapter 1 Welcome to Perforce: Installing and Upgrading

This chapter describes how to install a Perforce server or upgrade an existing installation.

Warning! If you're upgrading an existing installation to Release 2001.1 or later, see the notes in "Upgrading a Perforce Server" on page 16 before proceeding.

A short checklist of things to consider at installation time is offered, along with some basic security and administration tips. More detailed notes on administrative tasks are found in later chapters.

Windows Windows administrators will note that many of the examples in this book are based on the UNIX version of the Perforce server. In almost all cases, they are common to both Windows and UNIX installations; Perforce's behavior is generally the same regardless of whether executed in a UNIX shell or from the MS-DOS command line.

Where the UNIX and Windows versions of Perforce differ, a note to that effect will be made. For Windows-specific information, see "Perforce and Windows" on page 103.

OS X The material for UNIX also applies to Mac OS X.

Getting Perforce

Perforce requires at least two executables: the server (`p4d`), and any of the Perforce client programs (for instance, `p4` on UNIX, `p4.exe` or `p4win.exe` on Windows).

The programs are available from the Downloads page on the Perforce web site:

```
http://www.perforce.com/perforce/loadprog.html
```

Go to the web page, select the files for your platform, and save the files to disk.

Installing Perforce on UNIX

Although `p4` and `p4d` can be installed in any directory, on UNIX the Perforce client programs typically reside in `/usr/local/bin`, and the Perforce server is usually located either in `/usr/local/bin` or in its own server root directory. Perforce client programs can be installed on any machine that has TCP/IP access to the `p4d` host.

To limit access to the Perforce server files, we recommend that `p4d` be owned and run by a Perforce user account that has been created for that purpose.

Once you've downloaded `p4` and `p4d`, you need to do a few more things before you can use Perforce. Briefly:

- Download the files and make them executable,
- Create a root directory for the Perforce files,
- Provide a TCP/IP port to `p4d`,
- Provide the name of the Perforce server and the `p4d` port number to the Perforce client program(s), and
- Start the Perforce server (`p4d`).

Download the files and make them executable

On UNIX (or MacOS X), you'll also have to make the Perforce executables (`p4` and `p4d`) executable. After downloading the programs, use the `chmod` command to make them executable:

```
chmod +x p4
chmod +x p4d
```

Create a Perforce server root directory

Perforce stores all of its data in files and subdirectories of its own root directory, which can reside anywhere on the server system. This directory is called the *server root*. This directory should be owned by the account that runs `p4d`, and can be named anything at all. The only necessary permissions are `read`, `write`, and `execute` for the user who invokes `p4d`.

Since Perforce will store all submitted files under this directory, the size of the directory can become quite large. Disk space management is reviewed in "Installation and Administration Tips" on page 18 and described in more detail in "Disk space allocation" on page 92.

For security purposes, `read` and `write` access to the server root should be restricted to prevent anyone but the account owner from reading, modifying or even listing the actual depot files. To ensure that temporary files cannot be read by unauthorized users, set the `umask(1)` file creation-mode mask of the account owner to a value that will not permit other users to read the contents of the server root directory or its files.

You are strongly advised not to run `p4d` as `root` or any other privileged user. For more information, see the section entitled "UNIX: Run p4d as a non-privileged user" on page 22.

The environment variable `P4ROOT` should be set to point to the server root. Alternatively, the `-r` *`root_dir`* flag can be provided when `p4d` is started to specify a server root directory. The Perforce client programs never use this directory directly, and do not need

to know the value of P4ROOT; the p4d server is the only process which uses the P4ROOT environment variable.

Unlike P4ROOT, the environment variable P4PORT is used by both the Perforce server and Perforce client programs, and should be set on both. Its use is discussed in the next two sections.

Telling the Perforce server which port to listen to

The p4d server and Perforce client programs communicate with each other via TCP/IP. When p4d starts, it listens (by default) on port 1666. The Perforce client assumes (also by default) that its p4d server is located on host perforce, listening on port 1666.

If p4d is to listen on a different port, the port can be specified with the -p *port_num* flag when starting p4d (as in, p4d -p 1818), or the port can be set with the P4PORT environment or registry variable.

Telling Perforce client programs which port to talk to

Perforce client programs need to know the TCP/IP port on which the p4d server program is listening. The easiest way to do this under UNIX is to set each Perforce user's P4PORT environment variable to *host*:*port#*, where *host* is the name of the machine on which p4d is running, and *port#* is the port on which p4d is listening.

Examples:

If P4PORT is...	Then...
dogs:3435	The client program uses the p4d server on host dogs listening at port 3435.
x.com:1818	The client program uses the p4d server on host x.com listening at port 1818.

The definition of P4PORT can be shortened if the Perforce client program is running on the same host as p4d. In this case, only the p4d port number need be provided to the client. If p4d is running on a host named or aliased perforce, listening on port 1666, the definition of P4PORT for the client can be dispensed with altogether.

Examples:

If P4PORT is...	Then...
3435	The client program will use the p4d server on its local host listening at port 3435.
<not set>	The client program will use the p4d server on the host named or aliased perforce listening on port 1666.

If your p4d host is not named perforce, you can choose to simplify life somewhat for your Perforce users by setting perforce as an alias to the true host name in their workstations' /etc/hosts files, or by doing so via Sun's NIS or Internet DNS.

Starting the Perforce server

After p4d's P4PORT and P4ROOT environment variables have been set, p4d can be run in the background with the command:

```
p4d &
```

Although this command is sufficient to run p4d, other flags, for instance, those that control such things as error logging, checkpointing, and journaling, can be provided.

Example: *Starting a Perforce server.*

P4PORT can be overridden by starting p4d with the -p flag, and P4ROOT can be overridden by starting p4d with the -r flag. A journal file may be specified with the -J flag, and errors may be logged to a file specified with a -L flag. The startup command would then have this form:

```
p4d -r /usr/local/p4root -J /var/log/journal -L /var/log/p4err -p 1818 &
```

These flags (and others) are discussed in "Supporting Perforce: Backup and Recovery" on page 25. A complete list of server flags appears in the "Perforce Server (p4d) Reference" on page 111.

Stopping the Perforce server

If you are running Perforce 99.2 or above, use the command

```
p4 admin stop
```

to shut down the Perforce server. Only a Perforce superuser may use this command.

If you are running an earlier version of Perforce, you'll have to find the process ID of the p4d server and kill it manually from the UNIX shell. The use of kill -15 (SIGTERM) is preferable to kill -9 (SIGKILL), as the database could be left in an inconsistent state if p4d happened to be in the middle of updating a file when a SIGKILL signal was received.

Installing Perforce on Windows

Installation of Perforce on Windows is handled by the installer. You can get the installer by downloading it from the Downloads page of the Perforce web site.

The Perforce installer (`perforce.exe`) allows you to:

- Install Perforce client software ("User install").

  This option allows for the installation of `p4.exe` (the Perforce command line client), `p4win.exe` (P4Win, the Perforce Windows client), and `p4scc.dll` (Perforce's implementation of the Microsoft common SCM interface).

- Install Perforce as either a Windows server or service as appropriate. ("Administrator typical" and "Administrator custom" install).

  These options allow for the installation of both the Perforce client software as well as the Perforce Windows server (`p4d.exe`) and service (`p4s.exe`) executables.

  You can also use either of these options to automatically upgrade an existing Perforce server or service running under Windows.

  Under Windows 2000 or higher, you will need Administrator privileges to install Perforce as a service, and Power User privileges to install Perforce as a server.

- Uninstall Perforce: remove the Perforce server, service, and client executables, registry keys, and service entries. The database and depot files in your server root, however, will be preserved.

For more about installing on Windows, see "Using the Perforce installer" on page 103.

Terminology note: Windows services and servers

In most cases, it makes no difference whether the Perforce server program was installed on UNIX, as an NT service, or as an NT server. Consequently, the terms "Perforce server" and "`p4d`" are used interchangeably to refer to "the process which handles requests from Perforce clients". In cases where the distinction between an NT server and an NT service are important, the distinction is made.

On UNIX systems, there is only one Perforce "server" program (`p4d`) responsible for this back-end task. On Windows, however, this back-end program can be started either as an NT service (`p4s.exe`), which can be set to run at boot time, or as an NT server (`p4d.exe`), which is invoked from an MS-DOS prompt.

The Perforce service (`p4s.exe`) and the Perforce server (`p4d.exe`) executables are copies of each other; they are identical apart from their filenames. When run, they use the first three characters of the name with which they were invoked (either `p4s` or `p4d`) to determine their behavior. (For example, invoking copies named `p4smyservice.exe` or `p4dmyserver.exe` will invoke a service and a server, respectively.)

In most cases, you will want to install Perforce as a service, not a server. For a more detailed discussion of the distinction between the two options, see "Windows services vs. Windows servers" on page 105.

Starting and stopping Perforce on Windows

If you're running Perforce as a service under Windows, it will be started when the machine boots. You can configure it within the **Services** applet in the **Control Panel**.

If you're running Perforce as a server under Windows, invoke `p4d.exe` from an MS-DOS command prompt. The flags under Windows are the same as those under UNIX.

If you are running Perforce 99.2 or above, whether as a service or a server, use the command

```
p4 admin stop
```

to shut down the service or server. Only a Perforce superuser may use this command.

For older revisions of Perforce, you'll have to shut down services by using the **Services** applet in the **Control Panel**, and servers running in MS-DOS windows by typing CTRL-C in the window or clicking on the icon to **Close** the window.

While these options will work with both Release 99.2 and earlier versions of Perforce, they are not necessarily "clean", in the sense that the server or service is shut down abruptly. With the availability of the `p4 admin stop` command in 99.2, their use is no longer recommended.

Upgrading a Perforce Server

Whether your server is running on Windows or UNIX, you *must* back up your server (see "Backup Procedures" on page 31) as part of any upgrade process.

Warning! If you are upgrading to 2001.1, it is imperative that you read the notes pertaining to the 2001.1 upgrade.

Using old client programs with a new server

Although pre-98.1 Perforce client programs (`p4`, `p4.exe`, `p4win.exe`, and `p4scc.dll`) generally work with newer server versions with no trouble, some features in new server releases require client upgrades. In general, users with older client programs will still be able to use features available from the Perforce server at the client's release level, but will not be able to use the new server features offered by subsequent server upgrades.

Perforce's remote depot support is an exception: remote depot support is not guaranteed to work unless *all* of your Perforce servers are at or above Release 98.2.

Important Notes for 2001.1 and later

On small installations (installations with fewer than 1000 submitted changelists), the 2001.1 server automatically upgrades the underlying database for versions 98.2 and up.

On larger installations, you will have to upgrade the database manually. Although the new database will likely be smaller than the pre-2001.1 database, the upgrade process can be disk space-intensive. You will need approximately three times the size of the existing database to store the temporary files required by the upgrade.

Note If you are disk space-constrained, see the Release Notes for a more precise estimate of the amount of disk space required.

By turning off journaling during the upgrade (by setting `P4JOURNAL` to `off`), it may be possible to reduce the amount of disk space required for the upgrade. (Remember to turn journaling back on when the upgrade is complete!)

If you are upgrading from Release 97.3 or earlier to 2001.1, you will likely have to make an intermediate checkpoint. Please contact Perforce technical support for assistance before upgrading your server.

UNIX upgrades

To upgrade your current Perforce server to a newer version, your Perforce license file must be current. Expired licenses will not work with upgraded servers. (This is not a problem if you are running a two-user installation with no license.)

You *must* back up your server as described on "Backup Procedures" on page 31 as part of any upgrade process.

It is a good idea to run `p4 verify` as part of your upgrade. See "Verifying during server upgrades" on page 44 for details.

Warning! Upgrading to Release 2001.1 or later will require an upgrade of your database files. Downgrading thereafter will require restoration from backups.

If you wish to keep your pre-2001.1 server available as a fallback option when performing your 2001.1 upgrade, you should back up your entire server root (including the `db.*` files) after stopping the server.

Upgrading from UNIX Release 98.2 or later

If you have a valid license (or require no license) and are upgrading from Release 98.2 or later:

1. Download the new `p4d` executable for your platform
2. Stop the current instance of `p4d`
3. Make a checkpoint and back up your old installation

4. Install the new `p4d` in the desired location
5. Run `p4d -xu` to upgrade the database.

> **Note** If your server has fewer than 1000 changes, the upgrade will run automatically. Larger installations will require that `p4d -xu` be run manually.
>
> Either way, you must have sufficient disk space to complete the upgrade. The required amount is typically two to three times the size of the larger of the `db.have` or `db.integ` files.
>
> The `db.have` and `db.integ` files reside in your `P4ROOT` directory.

6. Restart the new `p4d` with your site's usual parameters.

Your users should then be able to use the new server.

Windows upgrades

On Windows, download the installer (`perforce.exe`) and follow the installation dialog.

The upgrade process on Windows is extremely conservative; if any error condition occurs during the upgrade, you will always be able to revert to using your pre-upgrade Perforce server or service.

> **Note** If your server has fewer than 1000 changes, the upgrade will run automatically. Larger installations will require that `p4d -xu` be run manually.
>
> Either way, you must have sufficient disk space to complete the upgrade. The required amount is typically two to three times the size of the larger of the `db.have` or `db.integ` files.
>
> The `db.have` and `db.integ` files reside in your `P4ROOT` directory.

If you have any questions or difficulties during an upgrade, contact Perforce technical support.

Installation and Administration Tips

Release and license information

Perforce servers are licensed according to how many users they will support.

This licensing information lives in a file called `license` in the server root directory. It is a plain text file supplied by Perforce Software. Without the `license` file, the Perforce server will limit itself to two users and two client workspaces.

Current licensing information may be viewed by invoking `p4d -V` from the server root directory or by specifying the server root directory either on the command line (`p4d -V -r server_root`) or in the `P4ROOT` environment variable.

When the server is running, the license information may also be viewed with `p4 info`.

Version information will be displayed when invoking `p4d -V` or `p4 -V`.

Observe proper backup procedures

Regular backups of your Perforce data are vital. The key concepts are:

- Make sure journaling is active,
- Create checkpoints regularly, and
- Use `p4 verify` regularly.

See "Supporting Perforce: Backup and Recovery" on page 25 for a full discussion of backup and restoration procedures.

Use separate physical drives for server root and journal

Whether installing on UNIX or Windows, it is usually advisable to have your `P4ROOT` directory (that is, the directory containing your database and versioned files) on a different physical drive than your journal file.

By storing the journal on a separate drive, you can be reasonably sure that if a disk failure corrupts the drive containing `P4ROOT`, it will *not* affect your journal file. The journal file can then be used to restore any lost or damaged metadata.

Further details are available in "Supporting Perforce: Backup and Recovery" on page 25.

Use protections and passwords

Until you define a Perforce superuser, every Perforce user is a Perforce superuser, and can run any Perforce command on any file. The administrator who installs Perforce should use:

```
p4 protect
```

to define a Perforce superuser as soon as possible after installing the Perforce server. For more information, see "Administering Perforce: Protections" on page 61

Furthermore, until your users have passwords defined, any user will be able to impersonate any other Perforce user, either with the `-u` flag or by setting `P4USER` to a

Perforce user's username. Proper use of Perforce passwords (as described in the *Perforce User's Guide*) can protect against this. See the *Perforce User's Guide* for details.

To set (or reset) a user's password, use `p4 passwd username` (as a Perforce superuser), and enter the new password for the user, or invoke `p4 user -f username` (also while as a Perforce superuser) and enter the new password into the user specification form. The former command will only work in release 99.1 or later; the latter command will work under all releases from 97.3 onwards.

The security-conscious Perforce superuser will use `p4 protect` to make sure that no access higher than `list` is granted to non-privileged users, and require that each user have a Perforce password.

Allocate disk space for anticipated growth

In general, you'll need sufficient space in your `P4ROOT` directory to hold your depot files (that is, the files created by your users), and an additional 0.5K per user per file to hold the data describing the files, their status, and their histories. As a rule of thumb, you also probably want at least enough disk space to hold three times the size of your present collection of versioned files.

For a more detailed example of a disk sizing estimate, see "Disk space allocation" on page 92.

Managing disk space after installation

All of Perforce's versioned files reside in subdirectories of the server root, as do its database files, and (by default) the checkpoints and journals. The stored versioned file depots are grow-only, and this can clearly present disk space problems on high use systems. The following approaches may be used to remedy this:

- Tell Perforce to store the journal file on a separate physical disk. Use the `P4JOURNAL` environment variable or `p4d -J` to specify the location of the journal file.
- Checkpoint on a daily basis to keep the journal file short.
- Compress checkpoints, or use the `-z` option to tell `p4d` to compress them while creating them.
- Use the `-jc prefix` option with the `p4d` command to write the checkpoint to a different disk. Alternately, use the default checkpoint files, but back up your checkpoints and then delete them from the root directory. Old checkpoints are needed when recovering from a crash, and if your checkpoint and journal files reside on the same disk as your depot, a hardware failure could leave you without the ability to restore your database.
- On UNIX systems, some or all of the depot directories may be relocated to other disks by using symbolic links. Creation of symbolic links and movement of depot files to other volumes should be done only while the Perforce server is not running.

- Due to the nature of the implementation of their access methods, the database files themselves may become internally unbalanced, resulting in them taking up more space than necessary. The database files can sometimes be reduced in size by recreating them from a checkpoint. This should be done only if the database files are more than about 10 times the size of the checkpoint in total. See “Checkpoints for Database Tree Rebalancing” on page 102.

Large filesystem support

Earlier versions of the Perforce server, as well as some operating systems, limit Perforce database files (the `db.*` files in the `P4ROOT` directory which contain your metadata) to 2GB in size.

The `db.have` file holds label contents and the list of files currently opened in client workspaces, and tends to grow the most quickly. If you anticipate any of your Perforce database files growing beyond the 2GB level, you should install the Perforce server on a platform with support for large files.

As of this writing, the following combinations of operating system and Perforce server revision will support database files larger than 2GB:

Operating System	OS version:	Perforce Server Revision
Tru64 UNIX (a.k.a. Digital UNIX, OSF/1)	All versions	98.2/5713 or higher
FreeBSD	All versions	98.2/5713 or higher
Windows NT, 2000	All versions, SP6 recommended for NT	98.2/8127 or higher
SGI IRIX 6.2	All versions	98.2/5713 or higher
SGI IRIX 5.3	Only with the SGI-supplied `xfs` upgrade	98.2/5713 or higher `xfs` OS upgrade required
Solaris	2.6 and higher	98.2/7488 *compiled for 2.6 or higher*

UNIX and NFS support

The Perforce server process has been tested and is supported on the Solaris 2.6 implementation of NFS. Because Perforce client programs never directly access the files in `P4ROOT`, the only process needing access to `P4ROOT` is the `p4d` server itself.

Consequently, under Solaris 2.6 or higher, you can store your journal, log, depot, and `db.*` files on NFS-mounted filesystems.

Some issues still remain regarding file locking on non-commercial implementations of NFS (for instance, Linux and FreeBSD). On these platforms, we recommend that you store your journal, log, depot, and `db.*` files on a drive local to the server machine, not on an NFS-mounted volume.

These issues affect only the PERFORCE server process (`p4d`). Perforce clients (such as the `p4` command-line client) have always been able to work with client workspaces on NFS-mounted drives, such as workspaces located in users' home directories.

Windows: Username and password required for network drives

By default, the Perforce service runs under the Windows local `System` account. Because Windows requires a real account name and password to access files on a network drive, if Perforce is installed as a service under Windows with `P4ROOT` pointing to a network drive, the installer will query for an account name and a password. The Perforce service will be configured with the supplied data and run as the specified user instead of `System`. (This account must have `Administrator` privileges on the machine.)

Although Perforce operates reliably with its root directory on a network drive, does so at a substantial performance penalty, as all writes to the database have to be performed over the network. For optimal performance, it is still best to install the Windows service to use local drives rather than networked drives.

For more information, see "Installing the Perforce service on a network drive" on page 107.

UNIX: Run p4d as a non-privileged user

While it is possible to run the Perforce server as `root`, it is highly inadvisable to do so. Sound administration practice demands that processes which don't require `root` access should never be run as `root`. For Perforce, this means that the owner of the `p4d` process should never be a privileged account.

Windows On Windows, directory permissions are set securely by default; when running as a server, the Perforce server root is accessible only to the user who invoked the server from the MS-DOS command line. When installed as a service, the files are owned by the `LocalSystem` account, and are accessible only to those with `Administrator` access.

A good way to manage a Perforce installation on UNIX is to create a UNIX userid for it (for example, "`perforce`") and (optionally) a UNIX group for it (for example, "`p4admin`"). The `umask(1)` command can be used to ensure that the server root (`P4ROOT`) and all files and directories beneath it are created as writable only by the UNIX user `perforce`, and (optionally) readable by members of the UNIX group `p4admin`.

The Perforce server (p4d), running as UNIX user perforce, can write to files in the server root, but none of your users will be able to overwrite its files. Access to read the files created by p4d (that is, the depot files, checkpoints, journals, and so on) can be granted to trusted users by making them members of the UNIX group p4admin.

Logging errors

The Perforce server's error output file can be specified with the -L flag to p4d, or can be defined in the environment variable P4LOG. If no error output file is defined, errors are dumped to p4d's standard error.

Although p4d tries to ensure that all error messages reach the user, if an error occurs and the client program disconnects before the error is received, p4d will also log these errors to its error output.

The Perforce server also has trace flags used for debugging purposes. See "Perforce server trace flags" on page 52 for details.

Case sensitivity issues

Whether your Perforce server is running on Windows or UNIX, if your site is involved in cross-platform development (i.e. Perforce clients on both Windows and UNIX machines), your users will still need to be made aware of certain details regarding case sensitivity issues. See "Case sensitivity and multi-platform development" on page 50 for details.

Tune for performance

Perforce is a relatively light consumer of network traffic and CPU power. The most important variables determining performance will be the efficiency of your server's disk I/O subsystem and the number of files referenced in any given user-originated Perforce operation.

For more detailed performance tuning information, see "Tuning Perforce for Performance" on page 91.

Chapter 2 Supporting Perforce: Backup and Recovery

The Perforce server stores two kinds of data: *versioned files* and *metadata*. Both are stored in the server's root directory.

- *Versioned files* are files submitted by Perforce users. Versioned files are stored in directory trees called *depots*. There is one subdirectory under the server's root directory for each depot in your Perforce installation. The versioned files for a given depot are stored in a tree of directories beneath this subdirectory.
- *Database files* store *metadata*, including changelists, opened files, client specs, branch specs, and other data concerning the history and present state of the versioned files. Database files appear as `db.*` files in the top level of the server root directory. Each `db.*` file contains a single, binary-encoded database table.

Backup and Recovery Concepts

Disk space shortages, hardware failures, and system crashes can corrupt any of the Perforce server's files. That's why the entire Perforce root directory structure (your versioned files and your database) should be backed up regularly.

As mentioned earlier, versioned files are stored in subdirectories beneath your Perforce server root, and can be restored directly from backups without any loss of integrity.

The files making up the Perforce database, on the other hand, may not have been in a state of transactional integrity at the moment they were copied to the system backups. Restoring the `db.*` files from system backups may result in an inconsistent database. The only way to guarantee the integrity of the database after it's been damaged is to reconstruct the `db.*` files from Perforce checkpoint and journal files.

- A *checkpoint* is a snapshot or copy of the database at a particular moment in time.
- A *journal* is a log that records updates made to the database since the last snapshot was taken.

The checkpoint file is often much smaller than the original database, and can be made smaller still by compressing it. The journal file, on the other hand, can grow quite large; it is truncated whenever a checkpoint is made, and the older journal is renamed. The older journal files can then be backed up offline, freeing up more space locally.

Both the checkpoint and journal are text files, and have the same format. A checkpoint and, if available, its subsequent journal, can restore the Perforce database.

Warning! Checkpoints and journals archive only the Perforce database files, ***not*** the files in the depot directories! You must always back up the depot files (your versioned files) with the standard OS backup commands after checkpointing.

Because the information stored in the Perforce database is as irreplaceable as your versioned files, checkpointing and journaling are an integral part of administering a Perforce server, and should be performed regularly.

Checkpoint files

A *checkpoint* is a file that contains all information necessary to recreate the metadata in the Perforce database. When you create a checkpoint, the Perforce database is locked, allowing you to take an internally consistent snapshot of that database.

Versioned files are backed up separately from checkpoints. This means that a checkpoint does *not* contain the contents of versioned files, and as such, ***you cannot restore any versioned files from a checkpoint.*** You can, however, restore all changelists, labels, jobs, and so on, from a checkpoint.

To guarantee database integrity upon restoration, the checkpoint must be as old as, or older than, the versioned files in the depot. This means that the database should be checkpointed, and the checkpoint generation must be complete, before the backup of the versioned files starts.

Regular checkpointing is important to keep the journal from getting too long. Making a checkpoint immediately before backing up your system is good practice.

Creating a checkpoint

Checkpoints are not created automatically; someone or something must run the checkpoint command on the Perforce server machine. You can create a checkpoint by invoking the `p4d` program with the `-jc` (journal-create) flag:

```
p4d -r root -jc
```

This can be run while the Perforce server (`p4d`) is running.

To make the checkpoint, `p4d` locks the database and then dumps its contents to a file named `checkpoint.`*`n`*, where *`n`* is a sequence number. Before it unlocks the database, `p4d` also copies the journal file to a file named journal.*`n-1`*, and then truncates the current journal. This guarantees that the last checkpoint (`checkpoint.`*`n`*) combined with the current journal (`journal`) will always reflect the full contents of the database at the time the checkpoint was created.

(The sequence numbers reflect the roll-forward nature of the journal; to restore databases to older checkpoints, match the sequence numbers. That is, the database reflected by `checkpoint.6` can be restored by restoring the database stored in `checkpoint.5` and rolling forward the changes recorded in `journal.5`. In most cases, you're only interested in restoring the current database, which is reflected by the highest-numbered `checkpoint.`*`n`* rolled forward with the changes in the current `journal`.)

You can specify a prefix for the checkpoint and journal filename by using the `-jc` option. That is, if you create a checkpoint with:

```
p4d -jc prefix
```

your checkpoint and journal files will be named `prefix.ckp.n`, or `prefix.jnl.n` respectively, where *`prefix`* is as specified on the command line and *`n`* is a sequence number. If no *`prefix`* is specified, the default filenames `checkpoint.n` and `journal.n` will be used.

Note The meaning of the argument to `-jc` changed in Release 99.2.

Prior to Release 99.2, the files created with `p4d -jc prefix` would have been `prefix.n` (for the checkpoint) and `journal.n` (for the old journal).

The behavior in 99.2 is a change from that in previous releases; if you have scripts which rely on the old behavior, you may have to modify them.

As of Release 99.2, if you need to take a checkpoint but are not on the machine running the Perforce server, you can create a checkpoint remotely with the `p4 admin` command. Use:

```
p4 admin checkpoint [prefix]
```

to take the checkpoint and optionally specify a *`prefix`* to the checkpoint and journal files. (You must be a Perforce superuser to use `p4 admin`.)

A checkpoint file may be compressed, archived, or moved onto another disk. At that time or shortly thereafter, the files in the depot subdirectories should be archived as well.

When recovering, *the checkpoint must be at least as old as the files in the depots.* (that is, the versioned files can be newer than the checkpoint, but not the other way around.) As you might expect, the shorter this time gap, the better.

You can set up an automated program to create your checkpoints on a regular schedule. Be sure to always check the program's output to ensure that the checkpoint creation was successful. The first time you need a checkpoint is not a good time to discover your checkpoint program wasn't working.

If the checkpoint command itself fails, contact Perforce Technical Support immediately. Checkpoint failure is usually a symptom of a resource problem (disk space, permissions, etc.) that can put your database at risk if not handled correctly.

Journal files

The *journal* is the running transaction log that keeps track of all database modifications since the last checkpoint. It's the bridge between two checkpoints.

If you have Monday's checkpoint and the journal that was collected from then until Wednesday, those two files (Monday's checkpoint plus the accumulated journal) contain the same information as a checkpoint made Wednesday. If a disk crash were to cause corruption in your Perforce database on Wednesday at noon, for instance, you could still restore the database even though Wednesday's checkpoint hadn't yet been made.

Warning! By default, the current journal file name is `journal` and it resides in the `P4ROOT` directory. However, if a disk failure corrupts that root directory, your journal file will be inaccessible too.

We strongly recommend that you set up your system so that the journal is written to a filesystem other than the `P4ROOT` filesystem. You can specify this from the command line, or set `P4JOURNAL` before starting the Perforce server to tell it where to write the journal.

To restore your database, you only need to keep the most recent journal file accessible, but it doesn't hurt to archive old journals with old checkpoints, should you ever need to restore to an older checkpoint.

Enabling journaling on Windows

For Windows installations, if you used the installer (`perforce.exe`) to install a Perforce server or service, journaling is turned on for you.

If you installed Perforce without the installer (for an example of when you might do this, see "Multiple Perforce services under Windows" on page 107), you do not have to create an empty file named `journal` in order to enable journaling under a manual installation on Windows.

Enabling journaling on UNIX

For UNIX installations, journaling is also automatically enabled.

If `P4JOURNAL` is left unset (and no location is specified on the command line), the default location for the journal is `$P4ROOT/journal`.

After enabling journaling

Be sure to create a new checkpoint with `p4d -jc` (and `-J` *`journalfile`* if required) immediately after enabling journaling. Once journaling is enabled, you'll need make regular checkpoints to control the size of the journal file. An extremely large current journal is a sign that a checkpoint is needed.

Every checkpoint after your first checkpoint starts a new journal file and renames the old one. The old `journal` is renamed to `journal.`*`n`*, (or *`prefix`*`.jnl.`*`n`* for Release 99.2 or later) where *n* is a sequence number, and a new `journal` file is created.

By default, the journal is written to the file `journal` in the server root directory (`P4ROOT`). Since there is no sure protection against disk crashes, the journal file and the Perforce server root should be located on different filesystems, ideally on different physical disk drives. The name and location of the journal can be changed by specifying the name of the journal file in the environment variable `P4JOURNAL`, or by providing the `-J` *`filename`* flag to `p4d`.

Warning! If you create a journal file with the `-J` *`filename`* flag, make sure that subsequent checkpoints use the same file, or the journal will not be properly renamed.

Whether you use `P4JOURNAL` or the `-J` *`journalfile`* option to `p4d`, the journal file name can be provided either as an absolute path, or as a path relative to the server root.

Example: *Specifying journal files*

Starting the server with:

```
$ p4d -r $P4ROOT -p 1666 -J /usr/local/perforce/journalfile
Perforce Server starting...
```

requires that you either checkpoint with:

```
$ p4d -r $P4ROOT -jc -J /usr/local/perforce/journalfile
Checkpointing to checkpoint.19...
Saving journal to journal.18...
Truncating /tmp/journalfile...
```

or set `P4JOURNAL` *to* `/usr/local/perforce/journal` *and use*

```
$ p4d -r $P4ROOT -jc
Checkpointing to checkpoint.19...
Saving journal to journal.18...
Truncating /tmp/journalfile...
```

If your `P4JOURNAL` *environment variable (or command-line specification) doesn't match the setting used when you started the Perforce server, the checkpoint is still created, but the journal is neither saved nor truncated. This is highly undesirable!*

Disabling journaling

To disable journaling, stop the server, remove the existing journal file (if it exists), set the environment variable `P4JOURNAL` to `off`, and restart `p4d` without the `-J` flag.

Versioned files

Your checkpoint and journal files are used to reconstruct the Perforce database files only. Your versioned files are stored in directories under the Perforce server root, and must be backed up separately.

Versioned file formats

Versioned files are stored in subdirectories beneath your server root. Text files are stored in RCS format, with filenames of the form *filename*,v. There is generally one RCS-format (,v) file per text file. Binary files are stored in full in their own directories named *filename*,d. Depending on the Perforce file type selected by the user storing the file, there may be one or more archived binary files in each *filename*,d directory. If more than one file resides in a *filename*,d directory, each one refers to a different revision of the binary file, and is named 1.*n*, where *n* is the revision number.

As of Release 99.2, Perforce also supports the AppleSingle file format for Macintosh. On the server, these files are stored in full, compressed, just like other binary files. They are stored in the Mac's AppleSingle file format; if need be, these files can be copied directly from the server root, uncompressed, and used as-is on a Macintosh.

Because Perforce uses compression in the depot files, a system administrator should not rely on the compressibility of the data when sizing backup media. Both text and binary files are either compressed by the Perforce server (denoted by the .gz suffix) before storage, or are stored uncompressed. At most installations, if any binary files in the depot subdirectories are being stored uncompressed, they were probably incompressible to begin with. (For example, many image, music, and video file formats are incompressible.)

Back up after checkpointing

In order to ensure that the versioned files reflect all the information in the database after a post-crash restoration, the db.* files must be restored from a checkpoint that is at least as old as (or older than) your versioned files. For this reason, you should create the checkpoint before backing up the versioned files in the depot directory or directories.

While your versioned files can be newer than the data stored in your checkpoint, it is in your best interest to keep this difference to a minimum; in general, you'll want your backup script to back up your versioned files immediately after successfully completing a checkpoint.

Backup Procedures

To back up your Perforce server, perform the following steps as part of your nightly backup procedure:

1. Verify the integrity of your server and add file signatures to any new files:

   ```
   p4 verify //...
   p4 verify -u //...
   ```

   You may wish to pass the `-q` (quiet) option to `p4 verify`. If called with the `-q` option, `p4 verify` will produce output only when errors are detected.

   The first command (`p4 verify`) recomputes the MD5 signatures of all of your archived files and compares them with those stored when `p4 verify -u` was first run on them. It also ensures that all files known to Perforce actually exist in the depot subdirectories; a disk-full condition that results in corruption of the database or archived files during the day can be detected by examining the output of these commands.

   The second command (`p4 verify -u`) updates the database with MD5 signatures for any new file revisions for which checksums have not yet been computed.

   By running `p4 verify -u` before the backup, you ensure that you create and store checksums for any files new to the depot since your last backup, and that these checksums are stored as part of the backup you're about to take.

   The use of `p4 verify` is optional, but is good practice not only because it allows you to spot any server corruption before a backup is made, but it also gives you the ability, following a crash, to detect whether or not the files restored from your backups are in good condition.

   Note If your site is very large, `p4 verify` may take some time to run, and you may wish to perform this step on a weekly basis rather than on a daily basis. For more about the `p4 verify` command, see "File verification by signature" on page 43.

2. Make a checkpoint by invoking `p4d` with the `-jc` (journal-create) flag, or by using the `p4 admin` command. Use one of:

   ```
   p4d -jc
   ```

   or (as of Release 99.2 or higher):

   ```
   p4 admin checkpoint
   ```

Because `p4d` locks the entire database when making the checkpoint, you do not generally have to stop your Perforce server during any part of the backup procedure.

Note If your site is very large (say, several GB of `.db` files), creating a checkpoint may take a considerable length of time.

Under such circumstances, you may wish to defer checkpoint creation and journal truncation until times of low system activity. You might, for instance, archive only the `journal` file in your nightly backup, and only create checkpoints and roll the journal file on a weekly basis.

If you are using the `-z` flag to create a gzip-compressed checkpoint, the checkpoint file will be named as specified. If you want the compressed checkpoint file to end in `.gz`, you should explicitly specify the `.gz` on the command line.

3. Ensure that the checkpoint has been created successfully before backing up any files. (After a disk crash, the last thing you want to discover is that the checkpoints you've been backing up for the past three weeks were incomplete!)

   You can tell that the checkpoint command has completed successfully by examining the error code returned from `p4d -jc`, or by observing the truncation of the current journal file.

4. Once the checkpoint has been created successfully, back up the checkpoint file, the old journal file, and your versioned files.

   (If you don't require an audit trail, you don't actually need to back up the journal. It is, however, usually good practice to do so.)

   Note There are rare instances (for instance, users obliterating files during backup, or submitting files on Windows during the file backup portion of the process) in which your depot files may change during the interval between the time the checkpoint was taken and the time at which the depot files get backed up by the backup utility.

   Most sites are affected by these issues; having the Perforce server available on a 24/7 basis is generally a benefit worth this minor risk, especially if backups are being performed at times of low system activity.

   If, however, the reliability of every backup is of paramount importance, consider stopping the Perforce server before checkpointing, and restarting it after the backup process has completed. Doing so will eliminate all risk of the system state changing during the backup process.

You never need to back up the `db.*` files. Your latest checkpoint and journal contain all the information necessary to re-create them. More significantly, a database

restored from `db.*` files is not guaranteed to be in a state of transactional integrity; a database restored from a checkpoint is.

Windows

On Windows, if you make your system backup while the Perforce server is running, you must ensure that your backup program doesn't attempt to back up the `db.*` files.

If you try to back up the `db.*` files with a running server, Windows locks them while the backup program backs them up. During this brief period, the Perforce server is unable to access the files; if a user attempts to perform an operation that would update the file, the server may fail.

If your backup software doesn't allow you to exclude the `db.*` files from the backup process, you should stop the server with `p4 admin stop` before backing up, and restart the server after the backup process is complete.

Recovery Procedures

If the database files become corrupted or lost, either because of disk errors, a hardware failure such as a disk crash, the database can be recreated with your stored checkpoint and journal.

There are many ways in which systems can fail; while this guide cannot address all of them, it can at least provide a general guideline for recovery from the two most common situations, specifically:

- corruption of your Perforce database only, without damage to your versioned files, and
- corruption to both your database and versioned files.

The recovery procedures for each failure are slightly different, and are discussed separately in the following two sections.

If you suspect corruption in either your database or versioned files, contact Perforce technical support.

Database corruption, versioned files unaffected

If only your database has been corrupted, (that is, your `db.*` files were on a drive that crashed, but you were using symbolic links to store your versioned files on a separate physical drive), you need only re-create your database.

You *will* need:

- The last checkpoint file, which should be available from the latest `P4ROOT` directory backup.

- The current journal file, which should be on a separate filesystem from your `P4ROOT` directory, and which should therefore have been unaffected by any damage to the filesystem where your `P4ROOT` directory was held.

You will *not* need:

- Your backup of your versioned files; if they weren't affected by the crash, they're already up to date.

To recover the database

1. Stop the current instance of `p4d`:

   ```
   p4 admin stop
   ```

   (You must be a Perforce superuser to use `p4 admin`.)

2. Rename (or move) the corrupt database ("`db.`") files:

   ```
   mv your_root_dir/db.* /tmp
   ```

   The corrupt `db.*` files aren't actually used in the restoration process, but it's safe practice not to delete them until you're certain your restoration was successful.

3. Invoke `p4d` with the `-jr` (journal-restore) flag, specifying your most recent checkpoint and current journal. If you explicitly specify the server root (`$P4ROOT`), the `-r $P4ROOT` argument must precede the `-jr` flag:

   ```
   p4d -r $P4ROOT -jr checkpoint_file journal_file
   ```

   This recovers the database as it existed when the last checkpoint was taken, and then apply the changes recorded in the journal file since the checkpoint was taken.

   Note If you're using the `-z` (compress) option to compress your checkpoints upon creation, you'll have to restore the uncompressed journal file separately from the compressed checkpoint.

   That is, instead of using:

   ```
   p4d -r $P4ROOT -jr checkpoint_file journal_file
   ```

   you'll use two commands:

   ```
   p4d -r $P4ROOT -z -jr checkpoint_file.gz
   p4d -r $P4ROOT -jr journal_file
   ```

   You must explicitly specify the `.gz` extension yourself when using the `-z` flag, and ensure that the `-r $P4ROOT` argument precedes the `-jr` flag.

Check your system

Your restoration is complete. See "Ensuring system integrity after any restoration" on page 37 to make sure your restoration was successful.

Your system state

The database recovered from your most recent checkpoint, after you've applied the accumulated changes stored in the current journal file, is up to date as of the time of failure.

After recovery, both your database and versioned files should reflect all changes made up to the time of the crash, and no data should have been lost.

Both database and versioned files lost or damaged

If both your database and your versioned files were corrupted, you need to restore both the database and your versioned files, and you'll need to ensure that the versioned files are no older than the restored database.

You *will* need:

- The last checkpoint file, which should be available from the latest `P4ROOT` directory backup.
- Your versioned files, which should be available from the latest `P4ROOT` directory backup.

You will *not* need:

- Your current journal file. The journal contains a record of changes to the metadata and versioned files that occurred between the last backup and the crash; because you'll be restoring a set of versioned files from a backup taken *before* that crash, the checkpoint alone contains the metadata useful for the recovery, and the information in the journal is of limited or no use.

To recover the database

1. Stop the current instance of `p4d`:

   ```
   p4 admin stop
   ```

   (You must be a Perforce superuser to use `p4 admin`.)

2. Rename (or move) the corrupt database ("`db.`") files:

   ```
   mv your_root_dir/db.* /tmp
   ```

   The corrupt `db.*` files aren't actually used in the restoration process, but it's safe practice not to delete them until you're certain your restoration was successful.

3. Invoke `p4d` with the `-jr` (journal-restore) flag, specifying *only* your most recent checkpoint:

   ```
   p4d -r $P4ROOT -jr checkpoint_file
   ```

This recovers the database as it existed when the last checkpoint was taken, but does not apply any of the changes in the journal file. (The `-r $P4ROOT` argument must precede the `-jr` flag.)

The database recovery without the roll-forward of changes in the journal file brings the database up to date as of the time of your last backup. In this scenario, you do not want to apply the changes in the journal file, because the versioned files you restored reflect only the depot as it existed as of the last checkpoint.

To recover your versioned files

4. After recovering the database, you then need to restore the versioned files according to your system's restoration procedures (for instance, the UNIX `restore(1)` command) to ensure that they are as new as the database.

Check your system

Your restoration is complete. See "Ensuring system integrity after any restoration" on page 37 to make sure your restoration was successful.

Note that files submitted to the depot between the time of the last system backup and the disk crash will not be present in the depot.

Note Although "new" files (submitted to the depot but not yet backed up) will not appear in the depot after restoration, it's possible (indeed, highly probable!) that at one or more of your users will have up-to-date copies of such files present in their client workspaces.

Your users can find such files by using Perforce to examine how files in their client workspaces differ from those in the depot. If they run:

```
p4 diff -se
```

...they'll be provided with a list of files in their workspace which differ from the files Perforce believes them to have. After verifying that these files are indeed the files you wish to restore, you may wish to have one of your users open these files for `edit` and submit them to the depot in a changelist.

Your system state

After recovery, your depot directories may not contain the newest versioned files. That is, files submitted after the last system backup but before the disk crash may have been lost.

- In most cases, the latest revisions of such files can be restored from the copies still residing in your users' client workspaces.
- In a case where *only* your versioned files (but *not* the database, which may have resided on a separate disk and remained unaffected by the crash) were lost, you may also be able to make a separate copy of your database and apply your journal to it in order to

examine recent changelists to track down files submitted between the last backup and the disk crash.

In either case, contact Perforce technical support for further assistance.

Ensuring system integrity after any restoration

After any restoration, it's wise to run `p4 verify` to ensure the versioned files are at least as new as the database:

```
p4 verify -q //...
```

This command verifies the integrity of the versioned files. The `-q` (quiet) option tells the command to only produce output on error conditions. Ideally, this command should produce no output.

If any versioned files are reported as `MISSING` by the `p4 verify` command, you'll know that there is information in the database concerning files that didn't get restored. The usual cause is that you restored from a checkpoint and journal made after the backup of your versioned files. (that is, that your backup of the versioned files was older than the database.)

If (as recommended) you've been using `p4 verify -u` to generate and store MD5 signatures for your versioned files as part of your backup routine, you can run `p4 verify` on the server after restoration to reassure yourself that your restoration was successful.

If you have any difficulties restoring your system after a crash, contact Perforce Technical Support for assistance.

Chapter 3 Administering Perforce: Superuser Tasks

This chapter describes both basic tasks associated with day-to-day Perforce administration as well as setting up advanced Perforce configurations, dealing with cross-platform development issues, migrating Perforce servers from one machine to another, and setting up remote and local depots.

Each of the tasks described in this chapter requires that you be a Perforce superuser as defined in the Perforce protections table. For more about controlling Perforce superuser access, and protections in general, see “Administering Perforce: Protections” on page 61.

Basic Perforce Administration

The following tasks commonly performed by Perforce administrators are covered:

- User operations, including resetting passwords, creating new users, disabling the automatic creation of new users, and cleaning up files left open by former users,
- Administrative operations, including obliterating files to reclaim disk space, editing previously submitted changelists, verifying server integrity, defining filetypes to fine-tune Perforce’s file type detection mechanism, and the use of the `-f` flag to force operations.

Resetting user passwords

From time to time, it is inevitable that users forget Perforce passwords. A Perforce superuser can set a new password for any user with:

- Release 99.1 and later:

  ```
  p4 passwd username
  ```

  (Perforce prompts the superuser for a new password for user *username*.)

- Pre-99.1 releases:

  ```
  p4 user -f username
  ```

  (The user specification form for *username* appears, and the new password may be entered.)

Creating new users

By default, Perforce creates a new user in its database whenever a command is issued by a username it hasn’t seen before.

You can use the `-f` (force) flag to create a new user as follows:

```
p4 user -f username
```

Fill in the form fields with the information for the user you wish to create.

The `p4 user` command also has an option (`-i`) to take its input from the standard input instead of the forms editor. If you wish to create a large number of new users at once, you can write a script which creates output in the same format as that used by the forms editor and then pipes each pre-generated "form" to `p4 users -i -f` *`username`*, where the *`username`* is also specified by a variable within your calling script.

Preventing creation of new users

As mentioned, Perforce's default behavior is to create a new user in its database whenever a command is issued by a username it hasn't seen before.

To prevent Perforce from automatically creating new users, all users must be explicitly listed in the protections table. The easiest way to ensure that this is the case is to put all users into a Perforce group, and to configure Perforce to only permit access to members of that group.

Example: *Setting up users in a group.*

A Perforce superuser wants to prevent the server from creating new users. He starts by setting up a group called `p4users` *for the three users currently at his site. He types:*

```
p4 group p4users
```

and fills in the form as follows:

```
# A Perforce Group Specification.
#
#  Group:       The name of the group.
#  MaxResults:  A limit on the data size of operations for users in
#               this group, or 'unlimited'.
#  Subgroups:   Other groups automatically included in this group.
#  Users:       The users in the group.  One per line.
Group:  p4users
MaxResults:     unlimited
MaxScanRows:    unlimited
Subgroups:
Users:
        edk
        lisag
        sarahm
```

He then uses `p4 protect` *to edit the protections table. The relevant line of the default protections table looks like this:*

```
write user * * //...
```

This grants `write` *permission to any* `user` *matching* `*` *(that is, to all users) from any host (the second* `*`*) in all areas of the depot (that is, to files in* `//...`*).*

After using `p4 group p4users` *to create the Perforce group* `p4users`*, he uses* `p4 protect` *to change this line in the protections table to read:*

```
write group p4users * //...
```

The replacement protection grants only `write` *access to users whose* `group` *matches* `p4users`*. Members of* `p4users` *may use Perforce from any host (*`*`*) and have write access to all areas of the depot (*`//...`*).*

As long as no other lines in the protections table grant permission to all users, all users are now defined within `p4 protect`*, and the server will no longer automatically create new user entries when new users attempt to access Perforce.*

For a more in-depth description of Perforce protections, see "Administering Perforce: Protections" on page 61.

Deleting obsolete users

Each user on the system consumes one Perforce license. You can free up licenses from unused users by deleting them.

```
p4 user -d username
```

You must first revert (or submit) any open files opened by a user before deleting that user. If you attempt to delete a user who has opened files, Perforce will display an error message to that effect.

Reverting files left open by obsolete users

If files have been left open by a nonexistent or obsolete user (for instance, a departing employee), a Perforce superuser can revert the files by deleting the client spec in which they were opened.

For example, if the output of `p4 opened` shows:

```
//depot/main/code/file.c#8 - edit default change (txt) by jim@stlouis
```

the "`stlouis`" client spec can be deleted with:

```
p4 client -d -f stlouis
```

Deleting a user's client spec automatically reverts all files opened by that client, and also removes that client's "have list". Note that it does *not* affect any files in the workspace actually used by that client; the files can still be accessed by other employees.

Reclaiming disk space by obliterating files

Warning! Use `p4 obliterate` with caution. This is the only command in Perforce that actually removes file data.

The depot is always growing, and this is not always desirable: a submit might have been performed incorrectly, creating hundreds of unneeded files, or perhaps there are simply a lot of old files around that are no longer being used.

Because `p4 delete` merely marks files as deleted in their head revisions, it can't be used to free up disk space on the server. This is where `p4 obliterate` can be useful. Superusers can use `p4 obliterate` *`filename`* to remove all traces of a file from a depot, making the file indistinguishable from one that never existed in the first place.

Note The purpose of a software configuration management system is to allow your site to maintain a history of which operations were performed on which files.

The `p4 obliterate` command defeats this purpose; as such, it is only intended to be used when cleaning up messes in the depot, and not as part of your normal software development process.

By default, `p4 obliterate` *`filename`* does nothing; it merely reports on what it would do. To actually destroy the files, use `p4 obliterate -y` *`filename`*.

If you need to destroy only one revision of a file (perhaps someone inadvertently stored some line art as a 20-megabyte uncompressed TIFF in place of its 500K-long compressed equivalent), specify only the desired revision number on the command line. For instance, to destroy revision #5 of a file, use:

```
p4 obliterate -y file#5
```

Revision ranges are also acceptable: To destroy revisions 5 through 7 of a file:

```
p4 obliterate -y file#5,7
```

Warning! If you mean to obliterate a revision range, be certain you've specified it properly. If you omit the specify revision range, ***all*** revisions of the file will be obliterated!

The safest way to use `p4 obliterate` is to use it ***without*** the `-y` (confirmation) flag until you're certain you've specified the files and revisions correctly.

The `p4 obliterate` command has one more flag: `-z`. When you branch a file from one area of the depot into another, a "lazy copy" is created - the file itself isn't copied, only a record that the branch has occurred. If, for some reason, you wish to undo the "lazy copy"

and create a new copy of the branched file's contents in your depot subdirectories, you could "obliterate" the lazy copy and create a new one by using `p4 obliterate -z filename`.

Unlike the `-y` flag, the `-z` flag *increases* disk space usage by removing the lazy copies. It's generally not a flag you'll use often, as its only use is to undo lazy copies in order to allow you to manually remove archive files without breaking any linked metadata pointing to the deleted files.

If a user sees the following error message while trying to access files:

```
Operation:user-sync
Librarian checkout path failed
```

where *path* is the path of a previously-obliterated file, the user has probably encountered a problem that resulted from an earlier use of `p4 obliterate` from an older (pre-98.2/10314) Perforce server. Contact Perforce technical support for a workaround.

Deleting changelists and editing changelist descriptions

You can use the `-f` (force) flag with `p4 change` to change the description or username of a submitted changelist. The syntax is `p4 change -f changenumber`. This presents the standard changelist form, but allows you to edit the change time, description, and/or username.

You can also use the `-f` flag to delete any submitted changelists that have been emptied of files with `p4 obliterate`. The full syntax is `p4 change -d -f changenumber`.

Example: *Updating changelist* `123` *and deleting changelist* `124`

Use `p4 change` *with the* `-f` *(force) flag:*

```
p4 change -f 123
p4 change -d -f 124
```

The `User:` *and* `Description:` *fields for change* `123` *are edited, and change* `124` *is deleted.*

File verification by signature

The `p4 verify filenames` command can be used to generate 128-bit MD5 signatures of each revision of the named files. A list of signatures stored by `p4 verify -u` can later be used to confirm proper recovery in case of a crash: if the signatures of the recovered files match the previously saved signatures, the files were recovered accurately.

To generate signatures and store them in the Perforce database, use the `-u` flag. Subsequent verifications will be compared against the stored signatures; if a new signature does not match the signature in the Perforce database for that file revision, Perforce adds the characters `BAD!` after the signature.

If you ever see a `BAD!` signature during a `p4 verify` command, your database or versioned files may have been corrupted, and you should contact Perforce Technical Support.

Because subsequent verifications can only be performed against previously stored signatures, the `p4 verify -u` command should be used regularly. A good strategy, for instance, might be to run `p4 verify` on a nightly basis before performing your system backups, proceeding with the backup only if the `p4 verify` reports no corruption. Generation and storage of new checksums (`p4 verify -u`) following a successful `p4 verify` could be performed nightly, or even weekly.

Verifying during server upgrades

It is also good practice to use `p4 verify` during server upgrades:

1. Before the upgrade, run:

   ```
   p4 verify -qu //...
   ```

   to generate the new checksums.

2. Take a checkpoint and copy the checkpoint and your versioned files to a safe place.

3. Perform the server upgrade.

4. After the upgrade, run:

   ```
   p4 verify -q //...
   ```

   to verify the integrity of your system.

Defining filetypes with p4 typemap

As of Release 2000.1, Perforce supports a new command: `p4 typemap`.

In previous releases, Perforce automatically determined if a file was of type `text` or `binary` based on an analysis of the first 1024 bytes of a file. If the high bit was clear in each of the first 1024 bytes, Perforce assumed it to be `text`; otherwise, it was `binary`. Although this default behavior could be overridden by the use of the `-t` *`filetype`* flag, it was easy to overlook this, particularly in cases where files' types were usually (but not always) detected correctly.

The `p4 typemap` command solves this problem by allowing system administrators to set up a table that links Perforce file types with file name specifications. If an entry in the typemap table matches an entry in the table, it overrides the file type that would otherwise be assigned by the Perforce client.

One common use for `p4 typemap` is for users dealing with Adobe PDF (Portable Document Format) files. Some PDF files start with a series of comment fields and textual data, and if the comments are sufficiently long, the files will be erroneously detected by

Perforce as being of type `text`. Similarly, files in RTF (Rich Text Format) format may sometimes be erroneously detected as `text`.

Perforce superusers may use `p4 typemap` to tell the Perforce server to regard all such files as `binary` by modifying the typemap table as follows:

```
Typemap:
        binary //....pdf
        binary //....rtf
```

The first three periods ("`...`") in the specification are a Perforce wildcard specifying that all files beneath the root directory are to be included as part of the mapping. The fourth period and the file extension specify that the specification applies to files ending in "`.pdf`" (or "`.rtf`").

For more information, see the `p4 typemap` page in the *Perforce Command Reference*.

Forcing operations with the -f flag

Certain commands allow the superuser to use the `-f` flag to force certain operations unavailable to ordinary users. This flag can be used with `p4 branch`, `p4 change`, `p4 client`, `p4 job`, `p4 label`, `p4 unlock` and `p4 user`. The usages and meanings of this flag are as follows:

Command	Syntax	Function
`p4 branch`	`p4 branch -f` *`branchname`*	Allows the modification date to be changed while editing the branch specification
	`p4 branch -f -d` *`branchname`*	Deletes the branch, ignoring ownership
`p4 change`	`p4 change -f [`*`changelist#`*`]`	Allows the modification date to be changed while editing the changelist specification
	`p4 change -f` *`changelist#`*	Allows the description field and username in a committed changelist to be edited
	`p4 change -f -d` *`changelist#`*	Deletes empty, committed changelists
`p4 client`	`p4 client -f` *`clientname`*	Allows the modification date to be changed while editing the client specification

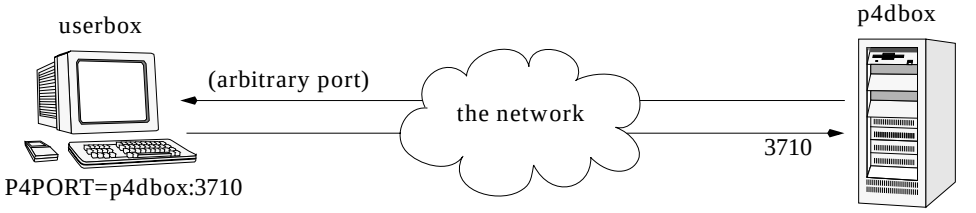
Command	Syntax	Function
	`p4 client -f -d` *clientname*	Deletes the client, ignoring ownership, even if the client has opened files
`p4 job`	`p4 job -f [`*jobname*`]`	Allows the manual update of read-only fields
`p4 label`	`p4 label -f` *labelname*	Allows the modification date to be changed while editing the label specification
	`p4 label -f -d` *labelname*	Deletes the label, ignoring ownership
`p4 unlock`	`p4 unlock -c` *changelist* `-f` *file*	Releases a lock (set with `p4 lock`) on an open file in a pending numbered changelist, ignoring ownership.
`p4 user`	`p4 user -f` *username*	Allows the update of all fields, ignoring ownership
	`p4 user -f -d` *username*	Deletes the user, ignoring ownership.

Advanced Perforce Administration

Running Perforce through a firewall

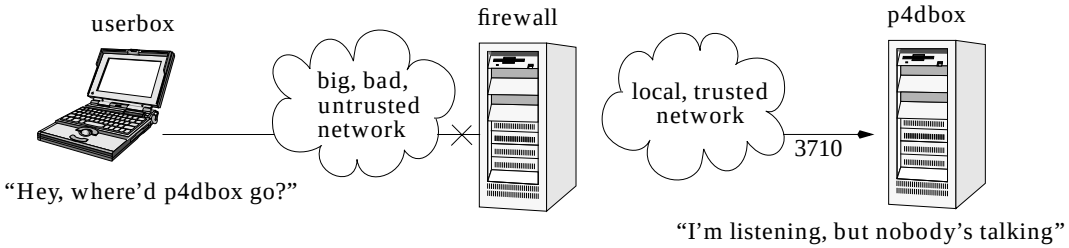
Perforce clients communicate with a Perforce server using TCP/IP. The server listens for connections at a specified port on the machine on which it's running, and clients make connections to that port.

The port on which the server listens is specified when the server is started. The number is arbitrary, so long as it does not conflict with any other networking services and is greater than 1024. The port number on the client machine is dynamically allocated.



A *firewall* is a network element which prevents any packets from outside a local (trusted) network from reaching that local network. This is done at a low level in the network protocol; any packets not coming from a trusted IP address are simply ignored.

In the following diagram, the Perforce client is on an untrusted part of the network. None of its connection requests reach the machine with the Perforce server. Consequently, the user running the client through the firewall is unable to use Perforce.



Secure shell

To solve this problem, you have to make the connection to the Perforce server from within the trusted network. This can be done securely using a package called *secure shell* (`ssh`).

Secure shell (`ssh`) is meant to be a replacement for the UNIX `rsh` (remote shell) command, which allows you to log into a remote system and execute commands on it. The "secure" part of "secure shell" comes from the fact that the connection is encrypted, so none of the data is visible while it passes through the untrusted network. With simple utilities like `rsh`, all traffic - even passwords - is unencrypted and visible to all intermediate hosts, creating an unacceptable security hazard.

Secure shell is available for free in source form for a multitude of UNIX platforms from `http://www.openssh.com`. This page also links to ports of `ssh` for OS/2 and Amiga, as well as commercial implementations for Windows and Macintosh from Data Fellows (`http://www.datafellows.com`) and SSH (`http://www.ssh.com`).

The OpenSSH FAQ (Frequently Asked Questions) can also be found online at the main site (`http://www.openssh.com/faq.html`).

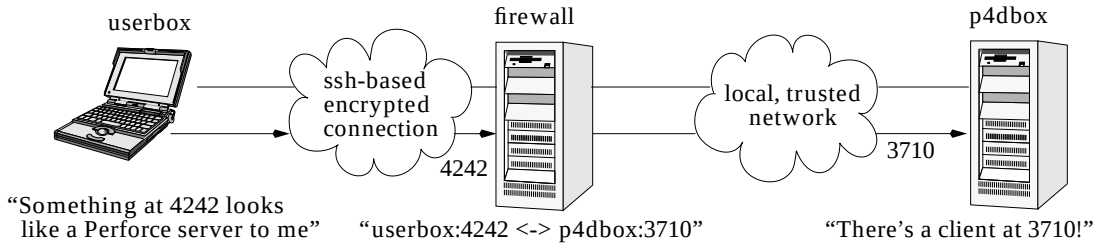
Solving the problem

Once you have `ssh` up and running, the simplest thing to do is to use it to log into the firewall machine and run the Perforce client from the firewall. While it has the advantage of simplicity, it's a poor solution: you typically want your client files accessible on your local machine, and of course, there's no guarantee that your firewall machine will match your development platform.

A good solution takes advantage of `ssh`'s ability to *forward* arbitrary TCP/IP connections. By using `ssh`, you can make your Perforce client appear as though it's connecting from the firewall machine over the local (trusted) network. In reality, your client remains on your

local machine, but all packets from your local machine are first sent to the firewall through the secure channel set up by `ssh`.

Suppose the Perforce server is on `p4dbox.bigcorp.com`, and the firewall machine is called `firewall.bigcorp.com`. In our example, we'll arbitrarily choose local port `4242`, and assume that the Perforce server is listening on port `3710`.



Packets ultimately destined for your client's port `4242` are first sent to the firewall, and `ssh` forwards them securely to your client. Likewise, connections made to port `4242` of the firewall machine will end up being routed to port `3710` of the Perforce server.

On UNIX, the `ssh` command on your own machine to set up and forward the TCP/IP connection would be:

```
ssh -L 4242:p4dbox.bigcorp.com:3710 firewall.bigcorp.com
```

At this point, it may be necessary to provide a password to log into `firewall.bigcorp.com`. Once the connection is established, `ssh` listens at port `4242` on the local machine, and forwards packets over its encrypted connection to `firewall.bigcorp.com`; the firewall then forwards them by normal channels to port `3710` on `p4dbox.bigcorp.com`.

All that remains is to tell the Perforce client to use port `4242` by setting the environment variable `P4PORT` to `4242`.

Normally, setting `P4PORT=4242` would normally indicate that we are trying to connect to a Perforce server on the local machine listening at port `4242`. In this case, `ssh` takes the role of the Perforce server. Anything a client sends to port `4242` of the local machine is forwarded by `ssh` to the firewall, which passes it to the real Perforce server at `p4dbox.bigcorp.com`. Since all of this is transparent to the Perforce client, it doesn't matter whether the client is talking to an instance of `ssh` that's forwarding traffic from port `4242` of the local machine, or if it's talking to a real Perforce server residing on the local machine.

The only glitch is that there's a login session you don't normally want on the firewall machine.

This can be solved by running

```
ssh -L 4242:p4dbox.bigcorp.com:3710 firewall.bigcorp.com -f sleep 9999999 -f
```

on the remote system.

This tells `ssh` on `firewall.bigcorp.com` to fork a long-running `sleep` command in the background after the password prompt. Effectively, this sets up the `ssh` link and keeps it up; there is no login session to terminate.

Finally, `ssh` can be configured to "do the right thing" so that it is unnecessary to type such a long command with each session. The Windows version of `ssh`, for instance, has a GUI to configure this.

One final concern: with port `4242` on the local machine now forwarded to a supposedly secure server, your local machine is part of the trusted network; it is prudent to make sure the local machine really *is* secure. The Windows version of `ssh` has an option to *only* allow local connections to the forwarded port, which is a wise precaution; your machine will be able to use port `4242`, but a third party's machine will be ignored.

Specifying IP addresses in P4PORT

Under most circumstances, your Perforce server's `P4PORT` setting consists solely of a port number.

If, however, you specify both an IP address *and* a port number in `P4PORT` when starting `p4d`, the Perforce server takes the IP address into account, and ignores requests from any IP addresses other than the one specified in `P4PORT`.

Although this isn't the default behavior, it can be useful. For instance, if you want to tell `p4d` to listen only to a specific network interface or IP address, you can make your Perforce server ignore all non-local connection requests by setting `P4PORT=localhost:`*`port`*.

Running from inetd on UNIX

Under a normal installation, the Perforce server is run on UNIX as a background process which waits for connections from clients. It is possible, however, to have `p4d` start up only when connections are made to it, using `inetd` and `p4d -i`.

If you wish to do this, add the following line to `/etc/inetd.conf`:

```
p4dservice stream tcp nowait username /usr/local/bin/p4d p4d -i -rp4droot
```

and add the following to `/etc/services`:

```
p4dservice nnnn/tcp
```

where:

- *`p4dservice`* is the service name you choose for this Perforce server
- `/usr/local/bin` is the directory holding your `p4d` binary

- *p4droot* is the root directory (P4DROOT) to use for this Perforce server (for example, /usr/local/p4d)
- *username* is the UNIX user name to use for running this Perforce server
- *nnnn* is the port number for this Perforce server to use

Note the "extra" p4d on the /etc/inetd.conf line must be there; inetd passes this to the OS as argv[0]. The first argument, then, is the -i flag, which causes p4d not to run in the background as a daemon, but rather to serve the single client connected to it on stdin/stdout. (This is the convention used for services started by inetd.)

This method is an alternative to running p4d from a startup script. It can also be useful for providing special services; for example, at Perforce, we have a number of test servers running on UNIX, each defined as an inetd service with its own port number.

There are caveats with this method:

- inetd may disallow excessive connections, so a script which invokes several thousand p4 commands, each of which spawns an instance of p4d via inetd may cause inetd to temporarily disable the service. Depending on your system, you may need to configure inetd to ignore or raise this limit.
- There is no easy way to disable the server, since the p4d executable is run each time; disabling the server requires modifying /etc/inetd.conf and restarting inetd.

Case sensitivity and multi-platform development

Early (pre-97.2) releases of the Perforce server treated all filenames, pathnames, and database entity names with case significance, whether the server was running on UNIX or Windows.

For example, //depot/main/foo.c and //depot/MAIN/FOO.C were treated as two completely different files. This caused problems where users on UNIX were connecting to a Perforce server running on Windows, because the filesystem underlying the server could not store files with the case-variant names submitted by UNIX users.

If you are running a pre-97.2 server on Windows, please contact support@perforce.com to discuss upgrading your server and database. In release 97.3, the behavior was changed, and only the UNIX server supports case-sensitive names. However, there are still some case-sensitivity problems which users can run into when sharing development projects across UNIX and Windows.

To summarize:

- The Perforce server on UNIX supports case-sensitive names.
- The Perforce server on Windows ignores case differences.
- Case is always ignored in keyword-based job searches, regardless of platform

Case-sensitive	UNIX server	Windows server
Pathnames and filenames	Yes	No
Database entities (clients, labels, etc.)	Yes	No
Job search keywords	No	No

To find out what platform your Perforce server runs on, use `p4 info`.

Perforce server on UNIX

If your Perforce server is on UNIX, and you have users on both UNIX and Windows, your UNIX users must be very careful not to submit files whose names differ only by case. Although the UNIX server can support these files, when Windows users sync their workspaces, they'll find files overwriting each other.

Conversely, Windows users will have to be careful to use case consistently in file and path names when adding new files. They may not realize that files added as `//depot/main/foo.c` and `//depot/MAIN/bar.c` will appear in two different directories in a UNIX user's workspace.

The UNIX Perforce server always respects case in client names, label names, branch view names, and so on. Windows users connecting to a UNIX server should be aware that the lowercased workstation names are used as the default names for new client workspaces. For examples, if a new user creates a client spec on a Windows machine named `ROCKET`, his client workspace is named `rocket` by default. If he later sets `P4CLIENT` to `ROCKET` (or `Rocket`), Perforce will tell him his client is undefined. He must set `P4CLIENT` to `rocket` (or unset it) to use the client workspace he defined.

Perforce server on Windows

If your Perforce server is running on Windows, your UNIX users must be aware that their Perforce server will store case-variant files in the same namespace.

For example, users who try something like this:

```
p4 add foo/file1
p4 add foo/file2
p4 add FOO/file3
```

should be aware that all three files will be stored in the same depot directory. The depot path and filenames assigned to the Windows server will be those first referenced. (In this case, the depot path name would be `foo`, and not `FOO`.)

Perforce server trace flags

You can turn on command tracing in the Perforce server by adding the `-v server=1` flag to the `p4d` startup command. Use `P4LOG` or the `-L` *`logfile`* flag to name a log file. For example:

```
p4d -r /usr/perforce -v server=1 -p 1666 -L /usr/perforce/logfile
```

Trace output appears in the specified log file, and shows the date, time, username, IP address, and command for each request processed by the server. Before turning on logging, you should make sure that you have adequate disk space.

Windows

Prior to Release 98.1, you could not set this trace flag when running Perforce as a service; you could set this flag (on Windows only) when running `p4d.exe` a server process from the MS-DOS command line.

As of Release 98.1, you can use the `p4 set` command to set `P4DEBUG` as a registry variable to "`server=1`" and thereby use this trace flag with Perforce installed as a service on Windows.

Prior to Release 97.3, the server trace flag was unavailable on any server platform.

In most cases, the Perforce server trace flags are useful only to administrators working with Perforce Technical Support to diagnose or investigate a problem.

Migrating to a new machine

The procedure for moving an existing Perforce installation from one machine to another depends on whether or not you're moving between machines

- of identical architecture,
- of different architectures using the same text file (CR/LF) format, or
- of different architecture *and* different text file format.

There are also additional considerations if the new machine has a different IP address or hostname.

The Perforce server stores two types of data under the Perforce root directory: *versioned files* and a *database* containing *metadata* describing those files. Your versioned files are the ones created and maintained by your users, and your database is a set of Perforce-maintained binary files holding the history and present state of the versioned files. In order to move a Perforce server to a new machine, both the versioned files and the database must be successfully migrated from the old machine to the new machine.

For more about the distinction between versioned files and database, as well as for an overview of backup and restore procedures in general, see "Backup and Recovery Concepts" on page 25.

Moving your versioned files and Perforce database

Between machines of the same architecture

If the architecture of the two machines is the same (e.g., SPARC/SPARC, x86/x86), the versioned files and database can be copied directly between the machines, and you only need to move the server root directory tree to the new machine. You can use `tar`, `cp`, `xcopy.exe`, or any other method. Copy everything in and under the `P4ROOT` directory - the `db.*` files (your database) as well as the depot subdirectories (your versioned files).

1. Back up your server (including a `p4 verify` before the backup) and take a checkpoint.
2. On the old machine, stop `p4d`.
3. Copy the contents of your old server root (`P4ROOT`) and all its subdirectories on the old machine into the new server root directory on the new machine.
4. Start `p4d` on the new machine with the desired flags.
5. Run `p4 verify` on the new machine to ensure that the database and your versioned files were transferred correctly to the new machine.

(Although the backup, checkpoint, and subsequent `p4 verify` are not strictly necessary in this case, it's always good practice to verify, checkpoint, and back up your system before any migration, and likewise to perform a subsequent verification after migration.)

Between different architectures using the same text format

If the internal data representation (big-endian vs. little-endian) convention differs between the two machines (e.g., Linux-on-x86/SPARC, NT-on-Alpha/NT-on-x86), but their operating systems use the same CR/LF text file conventions, you can still simply move the server root directory tree to the new machine.

Although the versioned files are portable across architectures, the database, as stored in the `db.*` files, is not. To transfer the database, you will need to create a checkpoint of your Perforce server on the old machine and use that checkpoint to recreate the database on the new machine. The checkpoint is a text file which can be read by a Perforce server on any architecture. For more details, see "Creating a checkpoint" on page 26.

After creating the checkpoint, you can use `tar`, `cp`, `xcopy.exe`, or any other method to copy the checkpoint file and the depot directories to the new machine. (You don't need to copy the `db.*` files, because they will be recreated from the checkpoint you took.)

1. On the old machine, use `p4 verify` to ensure that the database is in a consistent state.
2. On the old machine, stop `p4d`.
3. On the old machine, create a checkpoint:
   ```
   p4d -jc checkpointfile
   ```
4. Copy the contents of your old server root (`P4ROOT`) and all its subdirectories on the old machine into the new server root directory on the new machine.

   (To be precise, you don't need to copy the `db.*` files, just the checkpoint and the depot subdirectories. The `db.*` files will be recreated from the checkpoint. If it's more convenient to copy everything, then copy everything.)
5. On the new machine, if you copied the `db.*` files, be sure to remove them from the new `P4ROOT` before continuing.
6. Recreate a new set of `db.*` files suitable for your new machine's architecture from the checkpoint you created:
   ```
   p4d -jr checkpointfile
   ```
7. Start `p4d` on the new machine with the desired flags.
8. Run `p4 verify` on the new machine to ensure that the database and your versioned files were transferred correctly to the new machine.

Between Windows and UNIX

In this case, both the architecture of the system *and* the CR/LF text file convention may be different. You still have to create a checkpoint, copy it, and recreate the database on the new platform, but when you move the depot subdirectories containing your versioned files, you will also have to address the issue of the differing linefeed convention between the two platforms.

Depot subdirectories can contain both text and binary files. The text files (in RCS format, ending with "`,v`") and binary files (directories of individual binary files, each directory ending with "`,d`") will need to be transferred differently in order to translate the line endings on the text files while leaving the binary files unchanged.

As with all other migrations, be sure to run `p4 verify` after your migration.

Contact Perforce Technical Support for assistance when migrating a Perforce server from Windows to UNIX or vice-versa.

Changing the IP address of your server

If the IP address of the new machine is not the same as that of the old machine, you may need to update any IP-address-based protections in your protections table. See “Administering Perforce: Protections” on page 61 for information on setting protections for your Perforce server.

If you are a licensed Perforce customer, you will also need a new license file to reflect the new IP address. Contact Perforce technical support to obtain an updated license.

Changing the hostname of your server

If the hostname of the new machine serving Perforce is different from that of its predecessor, your users will need to change their `P4PORT` settings. If the old machine is being retired or renamed, consider setting an alias for the new machine to match that of the old machine, so that your users won't have to change their `P4PORT` settings.

Using Multiple Depots

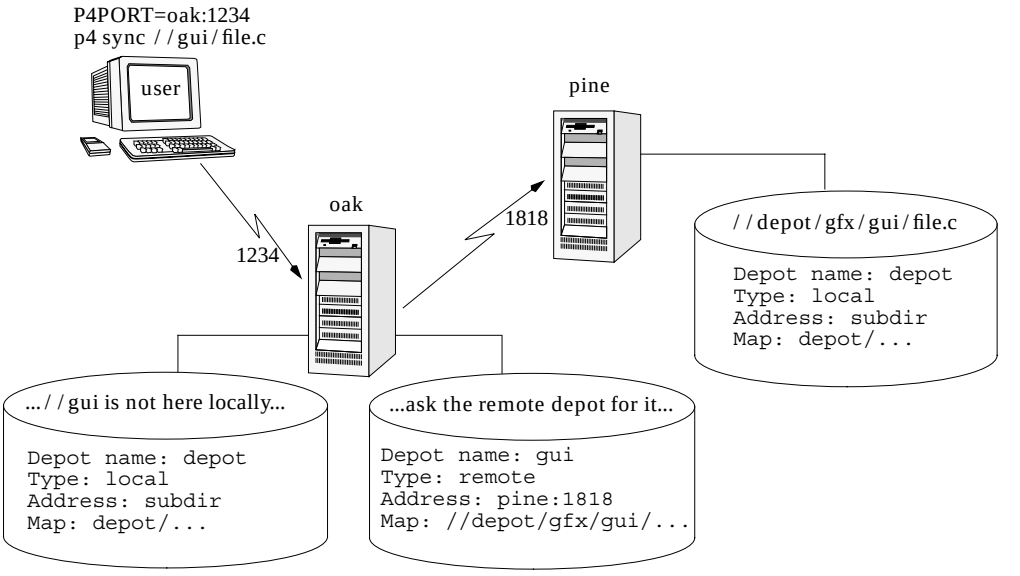
Just as Perforce servers can host multiple depots, Perforce client programs can access files from multiple depots. These other depots may reside within the Perforce server normally accessed by the Perforce client, or they may reside within other, *remote*, Perforce servers.

When using local depots, the user's Perforce client program communicates with the Perforce server specified by the user's `P4PORT` environment variable or equivalent setting.

When using remote depots, the user's Perforce client uses its default Perforce server as a proxy client to a second, *remote*, Perforce server. Because of this proxy behavior, the client doesn't need to know where the files are actually stored, and doesn't need direct access to the remote Perforce server.

The use of depots on remote servers (“remote depots”) is limited to read-only operations; a Perforce client may not add, edit, integrate into, or delete files that reside in depots on other servers. Depots sharing the same Perforce server as the client (“local depots”) are not subject to this limitation.

The following diagram illustrates how remote depots use a user's default Perforce server as a proxy.



Remote depot notes

The term “remote depot” is actually somewhat misleading. New Perforce customers tend to assume that if their users are geographically distributed, they need to set up separate Perforce installations (servers running `p4d`) and interconnect them with remote depot support.

This is not the case. Perforce is designed to cope with the latencies of large networks and inherently supports users with client workspaces at remote sites. A single Perforce installation is ready, out of the box, to support a shared development project, regardless of the geographic distribution of its contributors.

Remote depots are designed to support shared *code*, not shared *development*. They enable independent organizations with their own Perforce installations to view files and integrate changes from depots in other installations. Remote depots are not a generalized solution for load-balancing or network access problems.

When and when not to use remote depots

If you’re doing distributed development, you probably want to use a single Perforce installation, with all code in depots managed by one Perforce server. Partitioning joint

development projects into separate Perforce installations will not improve throughput, and usually only complicates administration.

If, however, you regularly import works from other organizations as part of your own organization's body of software, you may wish to consider using Perforce's remote depot features. This is what remote depots were designed for: facilitating the sharing of code, not development, across organizations.

Restrictions on remote depots

Users of remote depots encounter several restrictions:

- Access is restricted to read-only operations. You cannot submit files into a remote depot. You cannot edit files in remote depots. You can, however, create a branch in a local depot from files in a remote depot, and then integrate changes from the remote depot into the local branch. This integration is a one-way operation, as you cannot make changes in the local branch and integrate them back into the remote depot.
- Remote depots may be accessed only by Perforce servers running at the same release levels.
- Windows and UNIX servers are incompatible; you will get unpredictable results accessing remote depots on Windows from a UNIX server or vice versa.
- A Perforce server's metadata (information about client workspaces, changelists, labels, and so on) cannot be accessed using remote depots. Commands like `p4 client` only return specifications of entities defined in the local server's metadata.

In general, most of the restrictions associated with remote depots are either insignificant (lack of remote access to metadata) or beneficial (read-only access) if you're using remote depots for their intended purpose, namely the sharing of code, not development, between separate organizations.

Defining new depots

New depots (local or remote) in a server namespace are defined with the command `p4 depot` *`depotname`*. Depots may be defined as either *local* or *remote* depots.

Defining local depots

To define a new local depot (that is, a new depot in the current Perforce server namespace), call `p4 depot` with the new depot name, and edit only the `Map:` field in the resulting form. For example, to create a new depot called `book` with the files stored in the local Perforce server namespace in a root subdirectory called `book` (that is, `$P4ROOT/book`), enter the command `p4 depot book`, and fill in the resulting form as follows:

```
Depot:      book
Type:       local
Address:    subdir
Map:        book/...
```

Although you can set the `Map:` field to point to a depot directory other than one matching the depot name, there is rarely any advantage to be had by doing so, and it can make life confusing if you (as an administrator) ever need to work with the directories in the server root.

Defining remote depots

Defining a new depot on a remote Perforce server is only slightly more complicated than defining a local depot. Set the `Type:` to `remote`, provide the server's address in the `Address:` field, and set the `Map:` field to map into the remote depot namespace.

Example: *Defining a remote depot*

Lisa is working on a GUI *porting project. She and Ed are using different Perforce servers; his is on host* `pine`*, and it's listening on port* `1818`*. Lisa wants to grab Ed's* GUI *routines for her own use; she knows that Ed's color routine files are located on his Perforce server's single depot under the subdirectory* `graphics/GUI`*.*

Lisa's first step towards accessing Ed's files is to create a new depot. She'll call this depot `gui`*; she'd type* `p4 depot GUI` *and fill in the form as follows:*

```
Depot:      gui
Type:       remote
Address:    pine:1818
Map:        //depot/graphics/gui/...
```

This creates a remote depot called `gui` *on Lisa's Perforce server; this depot (*`gui`*) maps to Ed's* `depot`*'s namespace under its* `graphics/gui` *subdirectory.*

The Map: field

The `Map:` field is analogous to a client's view, except that the view may contain multiple mappings and the `Map:` field always contains a single mapping. This single client mapping format changes depending on whether the depot being defined is `local` or `remote`:

- For local depots, the mapping should contain a subdirectory relative to the file space of the Perforce server root directory. For example, `graphics/gui/...` maps to the `graphics/gui` subdirectory of the server root.

- For remote depots, the mapping should contain a subdirectory relative to the remote depot namespace. For example, `//depot/graphic/gui/...` would map to the `graphic/gui` subdirectory of the remote server depot named `depot`.

Note that the mapping subdirectory must always contains the "`...`" wildcard on its right side.

If you are unfamiliar with client views and mappings, please consult the *Perforce User's Guide* for general information about how Perforce mappings work.

Other depot operations

Naming depots

Depot names share the same namespace as branches, clients, and labels. For example, `//foo` refers uniquely to one of the depot `foo`, the client `foo`, the branch `foo`, or the label `foo`; you can't simultaneously have both a depot and a label named `foo`.

Listing depots

You can list all depots known to the current Perforce server with the `p4 depots` command.

Deleting depots

You can delete depots with `p4 depot -d` *`depotname`*.

To delete a depot, it must be empty; you must first obliterate all files in the depot with `p4 obliterate`.

For `local` depots, `p4 obliterate` deletes the versioned files as well as all their associated metadata. For `remote` depots, `p4 obliterate` erases *only* the locally held client and label records; the files and metadata still residing on the remote server remain intact.

Before using `p4 obliterate`, and *especially* if you're about to use it to obliterate all files in a depot, read and understand the warnings in "Reclaiming disk space by obliterating files" on page 42.

Limiting access from other servers

Remote depots are always accessed by a virtual user named `remote`, which does not consume a Perforce license. By default, all the files on any Perforce server may be accessed remotely.

To limit or eliminate remote access to a particular server, use `p4 protect` to set permissions for user `remote` on that server.

For example, to eliminate remote access to all files in all depots on a particular server, set the following permission on that server:

```
read user remote *  -//...
```

Since `remote` depots can only be used for `read` access, it is not necessary to remove `write` or `super` access.

Users working with multiple depots

If your users will be using remote depots as well as local depots, they should be aware of certain caveats regarding the behavior of files in remote depots as opposed to local depots, as described in the preceding sections.

The *Perforce User's Guide* contains detailed information for users who will be working with more than one depot.

Chapter 4 Administering Perforce: Protections

Perforce provides a protection scheme to prevent unauthorized or inadvertent access to the depot. The protections determine which Perforce commands can be run, on which files, by whom, and from which host. Protections are set with the `p4 protect` command.

When Should Protections Be Set?

Before `p4 protect` is run, every Perforce user is a superuser, and can access and change anything in the depot. The first time `protect` is invoked, a protections table is created that gives the invoking user superuser access from all hosts, and lowers everyone else's access level to `write` permission on all files from all hosts. Therefore, `protect` should be run as the concluding step of all new Perforce installations; the superuser can change the access levels as needed at any time.

The Perforce protections are stored in the `db.protect` file in the server root directory; if `p4 protect` is first run by an unauthorized user (or if you accidentally lock yourself out!) the depot can be brought back to its unprotected state by removing this file.

Setting Protections with "p4 protect"

The `p4 protect` form contains a single form field called `Protections:` that consists of multiple lines. Each line in `Protections:` contains subfields, and the table looks like this:

Example: *A sample protections table:*

```
Protections:
read        user       emily       *            //depot/elm_proj/...
write       group      devgrp      *            //...
write       user       *           195.3.24.*   -//...
write       user       joe         *            -//...
write       user       lisag       *            -//depot/...
write       user       lisag       *            //depot/doc/...
super       user       edk         *            //...
```

(The five fields may not line up vertically on your screen; they are aligned here for readability.)

The permission lines' five fields

Each line specifies a particular permission; each permission is defined by five fields.

The meanings of these fields are:

Field	Meaning
`Access Level`	Which access level is being granted: `list`, `read`, `open`, `write`, `review`, or `super`. These are described below.
`User/Group`	Does this protection apply to a user or a group? The value must be `user` or `group`.
`Name`	The user or group whose protection level is being defined. This field may contain the "`*`" wildcard. A "`*`" by itself grants this protection to everyone, "`*e`" grants this protection to every user (or group) whose username ends with an "`e`".
`Host`	The TCP/IP address of the host being granted access. This must be provided as the numeric address of the host in dotted quad notation (for instance, `192.168.41.2`). This field may contain the "`*`" wildcard. A "`*`" by itself means that this protection is being granted for all hosts. The wildcard can be used as in any string, so "`192.168.41.*`" would define access to any subnet within `192.168.41`, and "`*3*`" would refer to any IP address with a "`3`" in it. Since the client's IP address is provided by the Internet Protocol itself, this field provides as much security as is provided by the network.
`Files`	A file specification representing the files in the depot on which permissions are being granted. Perforce wildcards can be used in the specification. "`//...`" means all files in all depots.

Access levels

The access level is described by the first value on each line. The six access levels are:

Access Level	Meaning
`list`	Permission is granted to run Perforce commands that display file metadata, such as `p4 filelog`. No permission is granted to view or change the contents of the files.
`read`	The user(s) can run those Perforce commands that are needed to read files, such as `p4 client` and `p4 sync`. The `read` permission includes `list` access.

Access Level	Meaning
`open`	Grants permission to read files from the depot into the client workspace, and gives permission to open and edit those files. This permission does not allow the user to write the files back to the depot. `open` is similar to `write`, except that with `open` permission, users are not allowed to run `p4 submit` or `p4 lock`. The `open` permission includes `read` and `list` access.
`write`	Permission is granted to run those commands that edit, delete, or add files. The `write` permission includes `read`, `list`, and `open` access. This permission allows use of all Perforce commands except `protect`, `depot`, `obliterate`, and `verify`.
`review`	A special permission granted to review daemons. It includes list and read access, plus use of the `p4 review` command. Only review daemons require this permission.
`super`	For Perforce superusers; grants permission to run all Perforce commands. Provides `write` and `review` access plus the added ability to edit protections, create depots, obliterate files, and verify files.

Each Perforce command is associated with a particular minimum access level. For example, to run `p4 sync` on a particular file, the user must have been granted at least `read` access on that file. The access level required to run a particular command can usually be reasoned from knowledge of what the command does. For example, it is somewhat obvious that `p4 print` would require `read` access. For a full list of the minimum access levels required to run each Perforce command, see "How Protections are Implemented" on page 67.

Which users should receive which permissions?

The simplest method of granting permissions is to give `write` permission to all users who don't need to manage the Perforce system, and give `super` access to those who do, but there are times when this simple solution isn't sufficient.

`Read` access to particular files should be granted to users who don't ever need to edit those files. For example, an engineer might have `write` permission for source files, but have only `read` access to the documentation, while managers might be granted only `read` access to all files.

Because `open` access allows local editing of files, but doesn't allow these files to be written to the depot, `open` access is usually granted only in unusual circumstances. You might choose `open` access over `write` access when users are testing their changes locally, but when these changes should not be seen by other users. For instance, bug testers may want to change code in order to test theories as to why particular bugs occur, but these changes

would be for their own use, and would not be written to the depot. Perhaps a codeline has been frozen, and local changes are to be submitted to the depot only after careful review by the development team. In these cases, `open` access would be granted until the code changes have been approved, after which time the protection level would be upgraded to `write` and the changes submitted.

Default protections

Before `p4 protect` is invoked, every user has superuser privileges. When `p4 protect` is first run, two permissions are set by default. The default protections table looks like this:

```
write        user        *        *        //...
super        user        edk      *        //...
```

This indicates that `write` access is granted to all users, on all hosts, to all files. Additionally, the user who first invoked `p4 protect` (in this case, `edk`) is granted superuser privileges.

Interpreting multiple permission lines

The access rights granted to any user are defined by the union of mappings in the protection lines that match her user name and client IP address. (This behavior is slightly different when exclusionary protections are provided and is described in the next section.)

Example: *Multiple Permission Lines*

Lisa, whose Perforce username is `lisag`*, is using a client with the* IP *address* `195.42.39.17`*. The protections file reads as follows:*

```
read         user     *        195.42.39.17    //...
write        user     lisag    195.42.39.17    //depot/elm_proj/doc/...
read         user     lisag    *               //...
super        user     edk      *               //...
```

The union of the first three permissions apply to Lisa. Her username is `lisag`*, and she's currently using a client workspace on the host specified in lines 1 and 2. Thus, she can write files located in the depot's* `doc` *subdirectory, but can only read other files. Lisa tries the following:*

She types `p4 edit //lisag/doc/elm-help.1`*, and is successful.*

She types `p4 edit //lisag/READ.ME`*, and is told that she doesn't have the proper permission. She is trying to write to a file to which has only* `read` *access. She types* `p4 sync //lisag/READ.ME`*, and this command succeeds, as only* `read` *access is needed, and this is granted to her on line 1.*

Lisa later switches to another machine with IP *address* `195.42.39.13`. *She types* `p4 edit //lisag/doc/elm-help.1`, *and the command fails; when she's using this host, only the third permission applies to her, and she only has* `read` *privileges.*

Exclusionary protections

A user can be denied access to particular files by prefacing the fifth field in a permission line with a minus sign ("`-`"). This is useful for giving most users access to a particular set of files, while denying access to the same files to only a few users.

To use exclusionary mappings properly, it is necessary to understand some of their peculiarities:

- When an exclusionary protection is included in the protections table, the order of the protections is relevant: the exclusionary protection is used to remove any matching protections above it in the table.
- No matter what access level is provided in an exclusionary protection, all access levels for the matching files and IP addresses are denied. The access levels provided in exclusionary protections are irrelevant. The reasons for this counterintuitive behavior are described in the section "How Protections are Implemented" on page 67.

Example: *Exclusionary protections.*

Ed has used `p4 protect` *to set up protections as follows:*

```
write       user       *          *       //...
read        user       emily      *       //depot/elm_proj/...
super       user       joe        *       -//...
list        user       lisag      *       -//...
write       user       lisag      *       //depot/elm_proj/doc/...
```

The first permission looks like it grants write access to all users to all files in all depots, but this is overruled by later exclusionary protections for certain users.

The third permission denies Joe permission to access any file from any host. No subsequent lines grant Joe any further permissions; thus, Joe has been effectively locked out of Perforce.

The fourth permission denies Lisa all access to all files on all hosts, but the fifth permission gives her back `write` *access on all files within a specific directory. If the fourth and fifth lines were switched, Lisa would be unable to run any Perforce command.*

Granting Access to Groups of Users

Perforce *groups* simplify maintenance of the protections table. The names of users with identical access requirements can be stored in a single group; the group name can then be entered in the table, and all the users in that group receive the specified permissions.

Groups are maintained with `p4 group` and their protections assigned with `p4 protect`. Only Perforce superusers may use these commands.

Creating and editing groups

If `p4 group` *groupname* is called with a non-existent *groupname*, a new group named *groupname* is created. Calling `p4 group` with an existing *groupname* allows editing of the user list for this group.

The command `p4 group` *groupname* displays a form with two fields: `Group:` and `Users:`. The `Group:` field stores the group name, and cannot be edited; `Users:` is empty when the group is first created, and must be filled in. User names are entered under the `Users:` field header; each user name must be typed on its own line, and should be indented. A single user may be listed in any number of groups.

As of Release 99.2, groups can contain other groups, not just users. To add all users in a previously defined group to the group you're presently working with, include the group name in the `Subgroups:` field of the `p4 group` form. User and group names occupy separate namespaces, so groups and users can have the same names.

Groups and protections

To use a group with the `p4 protect` form, specify a group name instead of a user name in any line in the protections table, and set the value of the second field on the line to `group` instead of `user`. All the users in that group will be granted the specified access.

Example: *Granting access to Perforce groups.*

This protections table grants `list` *access to all members of the group* `devgrp`*, and* `super` *access to user* `edk`*:*

```
list        group       devgrp      *       //...
super       user        edk         *       //...
```

If a user belongs to multiple groups, one permission may override another, but the actual permissions granted to a specific user can be determined by replacing the names of all groups that a particular user belongs to with the user's name within the protections table, and applying the rules described earlier in this chapter.

Deleting groups

To delete a group, invoke

```
p4 group -d groupname
```

Alternately, invoke `p4 group groupname` and delete all the users from the group in the resulting editor form. The group will be deleted when the form is closed.

How Protections are Implemented

This section describes the algorithm that Perforce follows to implement its protection scheme. Protections can be used properly without reading this section, as the material here is provided to explain the logic behind the behavior described above.

Users' access to files is determined by the following steps:

- The command is looked up in the command access level table shown in "Access Levels Required by Perforce Commands" on page 68 to determine the minimum access level needed to run that command. In our example, `p4 print` is the command, and the minimum access level required to run that command is `read`.
- Perforce makes the first of two passes through the protections table. Both passes move up the protections table, bottom to top, looking for the first relevant line.

  The first pass determines whether or not the user is allowed to know whether or not the file exists. This search simply looks for the first line encountered that matches the user name, host IP address, and file argument. If the first matching line found is an inclusionary protection, then the user has permission to at least list the file, and Perforce proceeds to the second pass. Otherwise, if the first matching protection found is an exclusionary mapping, or if the top of the protections table is reached without a matching protection being found, then the user has no permission to even list the file, and will receive a message like `File not on client`.

Example: *Interpreting the order of mappings in the protections table.*

Suppose that our protections table is set as follows:

```
write       user        *       *       //...
read        user        edk     *       -//...
read        user        edk     *       //depot/elm_proj/...
```

If Ed runs `p4 print //depot/foo`*, Perforce examines the protections table bottom to top, and first encounters the last line. The files specified there don't match the file that Ed wants to print, so this line is irrelevant. The second-to-last line is examined next; this line matches Ed's user name, his* IP *address, and the file he wants to print; since this line is an exclusionary mapping, Ed isn't allowed to even list the file.*

- If the first pass is successful, a second pass is made at the protections table; this pass is the same as the first, except that access level is now taken into account.

  If an inclusionary protection line is the first line encountered that matches the user name, IP address, file argument, and has an access level greater than or equal to the access level required by the given command, then the user is given permission to run the command.

  If an exclusionary mapping is the first line encountered that matches according to the above criteria, or if the top of the protections table is reached without finding a matching protection, then the user has no permission to run the command, and will receive the message "`You don't have permission for this operation`".

Access Levels Required by Perforce Commands

The following table lists the minimum access level required to run each command. For example, since `p4 add` requires at least `open` access, `p4 add` can be run if `open`, `write` or `super` protections are granted.

Command	Access Level	Command	Access Level
`add`	`open`	`integrate` [d]	`open`
`admin`	`super`	`integrated`	`list`
`branch`	`open`	`job` [b]	`open`
`branches`	`list`	`jobs` [a]	`list`
`change`	`open`	`jobspec` [a b]	`super`
`changes` [a]	`list`	`label` [a]	`open`
`client`	`list`	`labels` [a b]	`list`
`clients`	`list`	`labelsync`	`open`
`counter` [c]	`review`	`lock`	`write`
`counters`	`list`	`obliterate`	`super`
`delete`	`open`	`opened`	`list`
`depot` [a b]	`super`	`passwd`	`list`
`depots` [a]	`list`	`print`	`read`
`describe`	`read`	`protect` [a]	`super`
`describe -s`	`list`	`reopen`	`open`
`diff`	`read`	`resolve`	`open`
`diff2`	`read`	`resolved`	`open`

Command	Access Level	Command	Access Level
dirs	list	revert	open
edit	open	review [a]	review
filelog	list	reviews [a]	list
files	list	set	list
fix [a]	open	submit	write
fixes [a]	list	sync	read
fstat	list	triggers	super
group [a b]	super	typemap	super
groups [a]	list	unlock	open
have	list	user [a b]	list
help	none	users [a]	list
info	none	verify	review
		where [a]	none

[a] This command doesn't operate on specific files. Thus, permission is granted to run the command if the user has the specified access to at least one file in the depot.
[b] The `-o` flag, which allows the form to be read but not edited, requires only `list` access.
[c] `list` access is required to view an existing counter's value; `review` access is required to change a counter's value or create a new counter.
[d] To run `p4 integrate`, the user needs `open` access on the target files and `read` access on the donor files.

Those commands that list files, such as `p4 describe`, will only list those files to which the user has at least `list` access.

Some of these commands (for instance, `p4 change`, when editing a previously submitted changelist) take a `-f` flag which can only be used by Perforce superusers. See "Forcing operations with the -f flag" on page 45 for details.

Chapter 5 Customizing Perforce: Job Specifications

Perforce's jobs feature allows changelists to be linked to enhancement requests, problem reports, and other user-defined tasks. Perforce also offers P4DTI (Perforce Defect Tracking Integration) as a way to integrate third-party defect tracking tools with Perforce. See "Working with third-party defect tracking systems" on page 79 for details.

The Perforce user's use of `p4 job` is discussed in the *Perforce User's Guide*. This chapter covers superuser modification of the jobs system.

Perforce's default jobs template has five fields for tracking jobs. These fields are sufficient for small-scale operations, but as projects managed by Perforce grow, the information stored in these fields may be insufficient. To modify the job template, use the `p4 jobspec` command. You must be a Perforce superuser to use `p4 jobspec`.

This chapter discusses the mechanics of altering the Perforce job template. Certain changes to the template are forbidden. Others are permissible, but are not recommended.

Warning! Improper modifications to the Perforce job template can lead to corruption of your server's database. Recommendations, caveats, and warnings about changes to job templates are summarized at the end of this chapter.

The Default Perforce Job Template

To understand how Perforce jobs are specified, we will examine the default Perforce job template. The examples that follow in this chapter are based upon modifications to the default Perforce job template.

A job created with the default Perforce job template has this format:

```
# A Perforce Job Specification.
#
#  Job:         The job name.  'new' generates a sequenced job number.
#  Status:      Either 'open', 'closed', or 'suspended'. Can be changed.
#  User:        The user who created the job. Can be changed.
#  Date:        The date this specification was last modified.
#  Description: Comments about the job.  Required.
Job:    new
Status: open
User:   edk
Date:   1998/06/03 23:16:43
Description:
        <enter description here>
```

The template from which this job was created can be viewed and edited with `p4 jobspec`. The default job specification template looks like this:

```
# A Perforce Job Specification.
#
#  Updating this form can be dangerous!
#  See 'p4 help jobspec' for proper directions.
Fields:
        101 Job word 32 required
        102 Status select 10 required
        103 User word 32 required
        104 Date date 20 once
        105 Description text 0 required
Comments:
        # A Perforce Job Specification.
        #
        # Job: The job name. 'new' generates a sequenced job number.
        # Status: Either 'open', 'closed', or 'suspended'. Can be changed.
        # User: The user who created the job. Can be changed.
        # Date: The date this specification was last modified.
        # Description: Comments about the job. Required.
Values:
        Status open/suspended/closed
Presets:
        Status open
        User $user
        Date $now
        Description $blank
```

The Job Template's Fields

There are six fields and field types in the `p4 jobspec` form. These fields define the template for all Perforce jobs stored on your server. The fields and field types are:

Field / Field Type	Meaning
`Fields:`	A list of fields to be included in each job. Each field consists of an ID#, a name, a datatype, a length, and a setting.
`Required:`	A list of fields for which values *must* be entered.
`Readonly:`	A list of fields, the default values of which cannot be changed by the user. Each field in this list requires a corresponding `Presets:` entry in the job specification.

Field / Field Type	Meaning
`Values:`	A list of fields whose datatype is `select`. For each `select` field, you must add a line containing the field's name, a space, and its list of acceptable values, separated by slashes.
`Presets:`	A list of fields and their default values. Values can be either literal strings or variables supported by Perforce.
`Comments:`	The comments that appear at the top of the `p4 job` form. These comments are also used by P4Win, the Perforce Windows client.

The Fields: field

The `p4 jobspec` field `Fields:` lists the fields to be tracked by your jobs, and specifies the order in which they appear on the `p4 job` form.

The default `Fields:` field lists these fields:

```
Fields:
        101 Job word 32 required
        102 Status select 10 required
        103 User word 32 required
        104 Date date 20 once
        105 Description text 0 required
```

Each field must be listed on a separate line, and is comprised of five separate descriptors:

Field Descriptor	Meaning
`ID#`	A unique integer identifier by which this field is indexed. After a field has been created and jobs entered into the system, the name of this field can change, but the data becomes inaccessible if the ID number changes. ID numbers must be between `101` and `199`.
`Name`	The name of the field as it should appear on the `p4 job` form.
`Data Type`	One of five datatypes (`word`, `text`, `line`, `select`, or `date`), as described in the next table.

Field Descriptor	Meaning
`Length`	The recommended size of the field's text box as displayed in P4Win, the Perforce Windows client. To display a text box with room for multiple lines of input, use a length of `0`; to display a single line, enter the `Length` as the maximum number of characters in the line. The value of this field has no effect on jobs edited from the Perforce command line, and is not related to the actual length of the values stored by the server.
`Persistence`	Determines whether a field is read-only, contains default values, is required, and so on. The valid values for this field are: • `optional`: field can take any value or can be deleted. • `default`: a default value is provided, but it can be changed or erased. • `required`: a default is given; it can be changed but the field can't be left empty. • `once`: read-only; the field is set once to a default value and is never changed. • `always`: read-only; the field value is reset to the default value when the job is saved. Useful only with the `$now` variable to change job modification dates, and with the `$user` variable to change the name of the user who last modified the job. In version 98.2 of Perforce, a field's persistence was specified in a very different way. If you have upgraded from 98.2, no conversion need be done; the old persistences will appear in the `p4 jobspec` form in the new template.

The five field datatypes are:

Field Type	Explanation	Example
`word`	A single word.	A userid: `edk`
`text`	A block of text that can span multiple lines.	A job's description
`line`	One line of text.	A user's real name: `Ed K.`

Field Type	Explanation	Example
`select`	One of a set of values. Each field with datatype `select` must have a corresponding line in the `Values:` field entered into the job specification.	A job's status. One of: `open/suspended/closed`
`date`	A date value.	The date and time of job creation: `1998/07/15:13:21:46`

The Presets: field

All fields with a persistence of anything other than `optional` require default values. To assign a default value to a field, create a line in the jobspec form under `Presets`, consisting of the field name to which you're assigning the default value. Any single-line string can be used as a default value.

Three variables are available for use as default values:

Variable	Value
`$user`	The Perforce user creating the job, as specified by the `P4USER` environment or registry variable, or as overridden with `p4 -u` *`username`* job.
`$now`	The date and time at the moment the job is saved.
`$blank`	The text `<enter description here>`. When users enter jobs, any fields in your jobspec with a preset of `$blank` must be filled in by the user before the job is added to the system.

The lines in the `Presets`: field for the standard jobs template are:

```
Presets:
        Status open
        User $user
        Date $now
        Description $blank
```

The Values: fields

You specify the set of possible values for any field of datatype `select` by entering lines in the `Values:` field. Each line should contain the name of the field, a space, and the list of possible values, separated by slashes.

In the default Perforce job specification, the `Status:` field is the only `select` field, and its possible values are defined as follows:

```
Values:
        Status open/suspended/closed
```

Note Prior to version 2000.1 of Perforce, the `Values:` and `Presets:` fields were specified differently.

If you have scripts that rely upon the old style of jobspecs, you might have to modify them. (Scripts that manipulate jobs, but not jobspecs, do not require modification.)

The Comments: field

The `Comments:` field supplies the comments that appear at the top of the `p4 job` form. Because `p4 job` does not automatically tell your users the valid values of `select` fields, which fields are required, and so on, your comments must tell your users everything they need to know about each field.

Each line of the `Comments:` field must be indented by at least one tab stop from the left margin, and must begin with the comment character `#`.

The comments for the default `p4 job` template appear as:

```
Comments:
        # A Perforce Job Specification.
        # Job: The job name. 'new' generates a sequenced job number.
        # Status: Either 'open', 'closed', or 'suspended'. Can be changed
        # User: The user who created the job. Can be changed.
        # Date: The date this specification was last modified.
        # Description: Comments about the job. Required.
```

If you administer a Perforce server and your users use P4Win, the Perforce Windows client, you must take extra care when writing your comments.

P4Win displays these comments in two ways:

- When the P4Win user creates or edits a job and presses the **Form Info** button in the job dialog box, a popup window displays the comments.
- As the (Windows) cursor moves over each field, the first line of the comment following the colon after the field name in the jobspec is displayed in a ToolTip. The remainder of each of these lines is displayed as the ToolTip for the field that matches the first word of the line. Only the first line of the comment is displayed.

  For instance, the ToolTip for the `Status:` field in the preceding jobspec reads:

  ```
  Either 'open', 'closed', or 'suspended'. Can be changed
  ```

Caveats, Warnings, and Recommendations

Although the material in this section has already been presented elsewhere in this chapter, it is important enough to bear repeating. Please follow the guidelines presented here when editing job specifications with `p4 jobspec`.

Warning! Please read and understand the material in this section before attempting to edit a job specification.

- After a field has been created and jobs have been entered, do not change the field's ID#. Any data entered in that field through `p4 job` will be inaccessible.
- Field names can be changed at any time. When changing a field's name, be sure to also change the fieldname in other `p4 jobspec` fields that reference this fieldname. For example, if you create a new field `106` named `severity` and subsequently rename it to `bug-severity`, then the corresponding line in the jobspec's `Presets:` field must be changed to `bug-severity` to reflect the change.
- The comments that you write in the `Comments:` field are the only way to let your users know the requirements for each field. Make these comments understandable and complete. These comments are treated specially in P4Win, the Perforce Windows client. For P4Win ToolTip compatibility, the first line of each field's comment should be understandable if read on its own.
- Leave the default fields `101` to `105` in the jobs system, and use `p4 jobspec` only to work with new fields. Do not change the names or types of fields `101` to `105` for any reason.
  - Field ID# `101`, the job's name, is required by Perforce and must not be deleted.
  - Field ID# `102`, the job's status, can be deleted; however, Perforce will subsequently be unable to update the status of jobs linked to particular changelists. If this field is present, Perforce will always set the value of this field to `closed` when a changelist containing this job is submitted, even if `closed` has been deleted from the list of possible values in the jobspec. This is highly undesirable behavior; do not change the name of field `102`.
  - Field ID# `105` is assumed to be a job description by Perforce clients. If present, it is also used by `p4 change` and `p4 submit` to describe the jobs fixed by the changelist.

Example: A Custom Template

The following example shows a more complicated jobspec and the resulting job form:

```
# A Custom Job Specification.
#
#  Updating this form can be dangerous!
#  See 'p4 help jobspec' for proper directions.
Fields:
        101 Job word 32 required
        102 Status select 10 required
        111 Type select 10 required
        112 Priority select 10 required
        113 Subsystem select 10 required
        114 Reported_by word 32 required
        115 Reported_date date 20 once
        105 Description text 0 required
Values:
        Type bug/sir/problem/unknown
        Status open/closed/suspended
        Priority A/B/C/unknown
        Subsystem server/gui/doc/mac/misc/unknown
Presets:
        Status open
        Type setme
        Priority unknown
        Subsystem setme
        Reported_by $user
        Reported_date $now
        Description $blank
Comments:
        # Custom Job fields:
        # Job:           Job number
        # Status:        Has the job been fixed: Acceptable values are
        #                'open', 'closed', or 'suspended'
        # Type:          The type of the job. Acceptable values are
        #                'bug', 'sir', 'problem' or 'unknown'
        # Priority:      How soon should this job be fixed?
        #                Values are 'a', 'b', 'c', or 'unknown'
        # Subsystem:     One of server/gui/doc/mac/misc/unknown
        # Reported_by:   Who's fixing the bug
        # Reported_date: When the bug was first entered
        # Description:   Textual description of the bug
```

Running `p4 job` against this jobspec displays the following job form:

```
# Custom Job fields:
# Job:           Job number
# Status:        Has the job been fixed: Acceptable values are
#                'open', 'closed', or 'suspended'
# Type:          The type of the job. Acceptable values are
#                'bug', 'sir', 'problem' or 'unknown'
# Priority:      How soon should this job be fixed?
#                Values are 'a', 'b', 'c', or 'unknown'
# Subsystem:     One of server/gui/doc/mac/misc/unknown
# Reported_by:   Who's fixing the bug
# Reported_date: When the bug was first entered
# Description:   Textual description of the bug

Job:    new

Status: open

Type:   setme

Priority:       unknown

Subsystem:      setme

Reported_by:    edk

Description:
        <enter description here>
```

The order of the listing under `Fields:` in the `p4 jobspec` form determines the order in which the fields appear to users in job forms; fields need not be ordered by numeric identifier.

Working with third-party defect tracking systems

With P4DTI, you can integrate Perforce with any third-party defect tracking or process management software.

Activity in your Perforce depot (enhancements, bug fixes, propagation of changes into release branches, and so forth) can be automatically entered into your defect tracking system by P4DTI. Conversely, issues and status entered into your defect tracking system (such as bug reports, change orders, work assignments) can be converted automatically to Perforce metadata for access by Perforce users.

P4DTI can be easily extended to other products; TeamShare and Bugzilla are the first two integrations published.

P4DTI is open source and available under a FreeBSD-like license.

Using P4DTI - Perforce Defect Tracking Integration

If you want to integrate Perforce with your in-house defect tracking system, or develop an integration with a third-party defect tracking system, P4DTI is probably the best place to start.

To get started with P4DTI, see the P4DTI product information page at:

```
http://www.perforce.com/perforce/products/p4dti.html
```

Available from this page are the TeamShare and Bugzilla implementations, an overview of the P4DTI's capabilities, and a kit (including source code and developer documentation) for building integrations with other products or in-house systems.

Building your own integration

Even if you don't use the P4DTI kit as a starting point, you can still use Perforce's job system as the interface between Perforce and your defect tracker. Depending on the application, the interface you set up will consist of one or more of the following:

- A trigger or script on the defect tracking system side that adds, updates, or deletes a job in Perforce every time a bug is added, updated, or deleted in the defect tracking system.

  The third-party system should generate the data and pass it to a script which reformats the data to resemble the form used by a manual (interactive) invocation of `p4 job`. The script can then pipe the generated form to a the standard input of a `p4 job -i` command.

  (The `-i` flag to `p4 job` allows `p4 job` to read a job form directly from the standard input, rather than using the interactive "form-and-editor" approach typical of user operations. Further information on automating Perforce with the `-i` option is available in the *Perforce Command Reference*.)

- A trigger on the Perforce side that checks changelists being submitted for any necessary bug fix information.

- A Perforce review daemon that checks successfully-submitted changelists for job fixes and issues the appropriate commands to update the corresponding data in your defect tracking system.

For more about triggers and review daemons, including examples, see "Scripting Perforce: Daemons and Triggers" on page 83.

Getting more information

In addition to the P4DTI-based TeamTrack and Bugzilla integrations, Perforce customers currently integrate Perforce with their own home-grown defect tracking systems, and with third-party systems such as Remedy, Scopus, and ClearTrack.

If you are interested in seeing what other Perforce users have done, visit the Perforce web site and examine the `perforce-user` mailing list archives, which are available under our Documentation page.

You may also wish to consider posting to `perforce-user` to ask if anyone has integrated Perforce with the third-party tool you're interested in, as someone may have already done all the setup work required to work with your system.

Chapter 6 Scripting Perforce: Daemons and Triggers

User-written scripts can enhance Perforce's functionality. There are three primary methods of scripting PERFORCE:

- *Wrappers* are scripts that call Perforce commands. Wrappers can be written in any scripting language, are usually tailored to your own site's needs, and are not discussed here.
- *Daemons* run at predetermined times, looking for changes to the Perforce metadata. When a daemon determines that the state of the depot has changed in some useful way, it runs other commands. For example, a daemon might look for newly submitted changelists and send email to users who have previously updated the files that were submitted in those changelists. Perforce provides a number of tools that make daemon-writing easier.
- *Pre-submit triggers* are scripts that Perforce runs whenever users attempt to submit files. A trigger script returns a value to Perforce that determines whether or not the submit should succeed. For example, you might write a script that watches for a particular executable file to be submitted; when this file is submitted, the trigger script might tell the submit to fail if the release notes file has not been updated within the same changelist.

This chapter assumes that you know how to write scripts.

Triggers

Warning! Although Perforce commands that only read data from the depot can be called in a trigger script, ***running PERFORCE commands that write data to the depot is dangerous and should be avoided.*** In particular, ***do not run the*** `p4 submit` command from within a trigger script.

A *pre-submit trigger* is a script called by Perforce when files that you've specified have been submitted. If the script exits with status 0, the submit continues. If it exits with a nonzero status, the submit fails. In the event of failure, the script's standard output is displayed as part of the error message returned to the user by the failed submit.

Triggers can be useful in many situations. Consider the following common uses:

- To validate submits above and beyond the Perforce protections mechanism. For example, it might ensure that Ed isn't allowed to submit file `foo` until Courtney has submitted file `bar`.

- To tell the submit to fail if file `foo` is not submitted in the same changelist as file `bar`.
- To ensure that every submit to a particular codeline fixes at least one job.

Triggers are created and edited with `p4 triggers`. Only the Perforce superuser can run this command. The `p4 triggers` form looks like this:

```
Triggers:
        relnotes_check  //depot/bld/...   "perl relcheck.pl %user%"
        verify_jobs     //depot/...       "python /usr/bin/job.py %change%"
```

Each line in the trigger table has three fields:

Field	Meaning
Trigger Name	The name of the trigger. The name can be any arbitrary string.
File Specification	A file specification in depot syntax. If a changelist contains any files that match this specification, the script will be run. Multiple file specifications can be linked to the same trigger by listing the trigger multiple times in the trigger table.
Script	The command that Perforce should run when a matching file is submitted. The submit will continue if the trigger script exits with 0, and fail otherwise. The script must be specified in a way that allows the Perforce server account to find the file. You can place the directory of the script in the `PATH` of the environment in which `p4d` is running, or specify the full path name of the script within the trigger table. This command should be quoted, and can take any or all of a number of variables as parameters. The most useful of these variables are: • `%user%`, which provides the Perforce name of the user who submitted the changelist; • `%changelist%`, (also abbreviated as `%change%`), the number of the changelist that's being submitted; and • `%client%`, the name of the client workspace from which the submit was run.

Example: *Creating a trigger*

The development group wants to make sure that whenever a `.exe` *file is submitted to the depot, the release notes for the program are submitted at the same time.*

You write a trigger script that takes a changelist number as its only argument, does a `p4 opened` *on the changelist, parses the results to find the files included in the changelist, and ensures that for every executable file that's been submitted, a* `RELNOTES` *file in the same directory has been submitted. If the changelist includes a* `RELNOTES` *file, the script terminates with an exit status of* `0`*; otherwise the exit status is set to* `1`*.*

The script written, you add it to the trigger table by editing the form displayed by `p4 trigger`*:*

```
Triggers:
        rnotes //depot/....exe   "/usr/bin/rnotetest.pl %changelist%"
```

Whenever an `.exe` *file is submitted, this trigger is run. If the script fails, it returns a nonzero exit status, and Perforce aborts the* `submit`*.*

Using triggers

Triggers are run in the order entered in the triggers table.

If you have multiple triggers associated with a file pattern, each will be run in the order in which it appears in the triggers table. If one of these triggers fails, no further triggers are executed.

Example: *Multiple triggers on the same file:*

In the next few examples, we're using `%change%` *as an abbreviation for* `%changelist%`*. Either form is acceptable, as they are interchangeable.*

All `*.c` *files must pass through the scripts* `check1.sh`*,* `check2.sh`*, and* `check3.sh`*:*

```
Triggers:
        check1     //depot/src/*.c  "/usr/bin/check1.sh %change%"
        check2     //depot/src/*.c  "/usr/bin/check2.sh %change%"
        check3     //depot/src/*.c  "/usr/bin/check3.sh %change%"
```

If any trigger fails (say, `check1.sh`*), the submit fails immediately and none of the subsequent triggers (that is,* `check2.sh` *and* `check3.sh`*) are called. Each time a trigger succeeds, the next matching trigger is run.*

If you have multiple filepatterns triggering the same script, you should create multiple triggers with separate names pointing to the same script. This is due to a limitation of Perforce.

Example: *A limitation: activating the same trigger for multiple filespecs:*

```
Triggers:
        bugcheck      //depot/*.c      "/usr/bin/checkit.pl %change%"
        bugcheck      //depot/*.h      "/usr/bin/checkit.pl %change%"
        bugcheck      //depot/*.cpp    "/usr/bin/checkit.pl %change%"
```

In this case, the detection of `foo.c` *in a changelist causes subsequent triggers with the same name (including the one intended for* `bar.h`*) to be ignored. The* `checkit.pl` *script only runs for* `*.c` *files. The* `*.h` *files and* `*.cpp` *files will not have the trigger applied.*

The workaround is to specify separately named triggers for each filespec: `*.c`*,* `*.h`*, and* `*.cpp`

```
Triggers:
        bugcheck1      //depot/*.c     "/usr/bin/checkit.pl %change%"
        bugcheck2      //depot/*.h     "/usr/bin/checkit.pl %change%"
        bugcheck3      //depot/*.cpp   "/usr/bin/checkit.pl %change%"
```

In this case, the `bugcheck1` *trigger runs on the* `*.c` *files, the* `bugcheck2` *trigger runs for the* `*.h` *files, and* `bugcheck3` *runs on the* `*.cpp` *files.*

Some trigger scripts need to know the files that are included in the changelist. Since `p4d` can only pass 1K of data to a trigger script, the file list can't be passed via the trigger. Use `p4 opened -ac` *`changelist#`* in your trigger scripts to get the list of files for the changelist number provided as an argument. The actual contents of the files are not accessible from within the trigger script, since the files are not stored in the depot until the submit completes.

Before `p4 submit` runs the trigger script, it creates the changelist, assigns it a number, and locks the files in the changelist. After a trigger script completes, `p4 submit` may run subsequent trigger scripts that cause the submit to fail. For this reason, trigger scripts should not take any actions that assume the submit will succeed. Trigger scripts are meant primarily for changelist validation; if you need to take particular actions based on the success of a submit, use a daemon.

Note In order to use triggers, the server (`p4d`) must be able to "fork", or spawn off processes to run the triggers. This is the default configuration of Perforce.

If you invoke `p4d` with the `-f` (run in foreground without forking) option, however, you will not be able to use triggers until you restart the server without the `-f` option.

Triggers and security

Warning! Because triggers are spawned by the `p4d` process, `p4d` should never be run as `root` on UNIX systems.

Triggers and Windows

By default, the Perforce service runs under the Windows local `System` account.

Because Windows requires a real account name and password to access files on a network drive, if the trigger script resides on a network drive, you must configure the service to use a real userid and password to access the script.

For details, see "Installing the Perforce service on a network drive" on page 107.

Daemons

Daemons are processes that are called periodically or run continuously in the background. Daemons that use Perforce usually work by examining the server metadata as often as needed and taking action as often as necessary. Typical daemon applications include:

- A change review daemon that wakes up every ten minutes to see if any changelists have been submitted to the production depot. If any changelists have been submitted, the daemon sends email to those users who have "subscribed" to any of the files included in those changelists. The message informs them that the files they're interested in have changed.
- A jobs daemon that generates a report at the end of each day to create a report on open jobs. It shows the number of jobs in each category, the severity each job, and more. The report is mailed to all interested users.
- A Web daemon that looks for changes to files in a particular depot subdirectory. If new file revisions are found there, they are synced to a client workspace that contains the live web pages.

Daemons can be used for almost any task that needs to occur when Perforce metadata has changed. Unlike triggers, which are used primarily for submission validation, daemons can also be used to write information (that is, submit files) to a depot.

Perforce's change review daemon

The change review daemon described above can be downloaded from `http://www.perforce.com/perforce/loadsupp.html`. It runs under Python, which can be retrieved from `http://www.python.org/`. Before running the script, please be sure to read and follow the configuration instructions included in the script itself.

The change review daemon looks at the files included in each newly submitted changelist and emails those users who have "subscribed" to any of the files included in the changelist, letting those users know that the file(s) they're interested in have changed.

Users subscribe to files by calling `p4 user` and entering filepatterns in the `Reviews:` field of the resulting form:

```
User:     sarahm
Email:    sarahm@elmco.com
Update:   1997/04/29 11:52:08
Access:   1997/04/29 11:52:08
FullName: Sarah MacLonnogan
Reviews:
        //depot/doc/...
        //depot.../README
```

Users should enter their email addresses in the `Email:` field, and enter any number of filepatterns corresponding to the files in which they're interested into the `Reviews:` field. The daemon reports changes to these files to the users.

By including the special path `//depot/jobs` in the `Reviews:` field, users can also receive mail from the Perforce change review daemon whenever job data is updated.

The change review daemon implements the following scheme:

1. `p4 counter` is used to read and change a variable, called a *counter*, in the Perforce metadata. The counter used by this daemon, `review`, stores the number of the latest changelist that's been reviewed.
2. The Perforce depot is polled for submitted, unreviewed changelists with the `p4 review -t review` command.
3. `p4 reviews` generates a list of reviewers for each of these changelists.
4. The Python mail module mails the `p4 describe` changelist description to each reviewer.
5. The first three steps are repeated every three minutes, or at some other interval configured the time of installation.

The command used in the fourth step (`p4 describe`) is a straightforward reporting command. The other commands (`p4 review`, `p4 reviews`, and `p4 counter`) are used almost exclusively by review daemons.

Creating other daemons

You can use `p4review.py` as a starting point to create your own daemons, changing it as needed. As an example, another daemon might upload Perforce job information into an external bug tracking system after changelist submission. It would use the `p4 review`

command with a new review counter to list new changelists, and use `p4 fixes` to get the list of jobs fixed by the newly submitted changelists. This information might then be fed to the external system, notifying it that certain jobs have been completed.

If you write a daemon of your own and would like to share it with other users, you can submit it into the Perforce Public Depot. For more information, go to `http://www.perforce.com` and follow the "Perforce Public Depot" link.

Commands used by daemons

Certain Perforce commands are used almost exclusively by review daemons.

These commands are:

Command	Usage
`p4 counter` *`name`* [*`value`*]	When a *`value`* argument is not included, `p4 counter` returns the value of the variable name. When a *`value`* argument appears, `p4 counter` sets the value of the variable name to *`value`*. Requires at least `review` access to run. ***WARNING:*** The review counters named `journal`, `job`, and `change` are used internally by Perforce; ***use of any of these three names as review numbers could corrupt the Perforce database.*** For Release 99.2 and above, Perforce will not let you change the values of `journal`, `job`, and `change`.
`p4 counters`	List all counters and their values.
`p4 review -c` *`changelist#`*	For all changelists between *`changelist#`* and the latest submitted changelist, this command lists the changelists' numbers, creators, and creators' email addresses. Requires at least `review` access to run.

Command	Usage
`p4 reviews -c changelist# filespec`	Lists all users who have subscribed to review the named files or any files in the specified changelist. It is hard to imagine any use for this command outside of our own change review daemon.
`p4 changes -m 1 -s submitted`	Output a single line showing the changelist number of the last submitted changelist, as opposed to the highest changelist number known to the Perforce server.

Daemons and counters

If you're writing a change review daemon or other daemon that deals with submitted changelists, you may also wish to keep track of the changelist number of the last *submitted* changelist, which is the second field in the output of a `p4 changes -m 1 -s submitted` command.

This is *not* the same as the output of `p4 counter change`. The last changelist number known to the Perforce server (the output of `p4 counter change`) includes pending changelists created by users, but not yet submitted to the depot.

Scripting and buffering

Depending on your platform, the output of individual `p4` commands may be fully-buffered (output flushed only after a given number of bytes generated), line-buffered (as on a tty, one line sent per linefeed), or unbuffered.

In general, stdout to a file or pipe is fully-buffered, and stdout to a tty is line-buffered. If your trigger or daemon requires line-buffering (or no buffering), you can disable buffering by supplying the `-v0` debug flag to the `p4` command in question.

If you're using pipes to transfer standard output from a Perforce command (with or without the `-v0` flag), you may also experience buffering issues introduced by the kernel, as the `-v0` flag can only unbuffer the output of the command itself.

Chapter 7 Tuning Perforce for Performance

Your Perforce server should normally be a light consumer of system resources. As your installation grows, however, you may wish to revisit your system configuration to ensure that it is configured for optimal performance.

The following chapter briefly outlines some of the factors that can affect the performance of a Perforce server, provides a few tips on diagnosing network-related difficulties, and offers some suggestions on decreasing server load for larger installations.

Tuning for Performance

The following variables can affect the performance of your Perforce server.

Memory

Server performance is highly dependent upon having sufficient memory. Two bottlenecks are relevant: the first can be avoided by ensuring that the server doesn't page when running large queries, and the second by ensuring that the `db.rev` table (or at least as much of it as practical) can be cached in main memory.

- Determining memory requirements for large queries is fairly straightforward: the server requires about 1KB / file of RAM to avoid paging; 10,000 files will require 10MB of RAM.
- To cache `db.rev`, the size of the `db.rev` file in an existing installation can be observed and used as an estimate. New installations of Perforce can expect `db.rev` to require about 150-200 bytes per revision, and roughly 3 revisions per file, or about 0.5KB of RAM per file.

Thus, if there is 1.5KB of RAM available per file, or 150MB for 100,000 files, the server will not page, even when performing an operation involving all files. It is still possible that multiple large operations will be performed simultaneously and thus require more memory to avoid paging. On the other hand, the vast majority of operations will only involve a small subset of files.

For most installations, a system with enough RAM for 1.5KB per file in the depot will suffice.

Filesystem performance

Perforce is judicious with regards to its use of disk I/O; its metadata is well-keyed and accesses are mostly sequential scans of limited subsets of the data.

The only disk-intensive activity is file check-in, where the Perforce server must write and rename files in the archive. Server performance depends heavily upon the operating system's filesystem implementation, and in particular, whether directory updates are synchronous.

Although Perforce does not recommend any specific filesystem, Linux servers are generally fastest (owing to Linux's asynchronous directory updating), but may have poor recovery if power is cut at the wrong time. The BSD filesystem (also used in Solaris) is relatively slow, but much more reliable. NTFS performance falls somewhere in between these two ranges. The filesystems used by IRIX and OSF have demonstrated an excellent combination of both speed and robustness.

Performance in systems where database and versioned files are stored on NFS-mounted volumes is typically dependent on the implementation of NFS in question and/or the underlying storage hardware. Perforce has been tested and is supported under the Solaris implementation of NFS.

Under Linux and FreeBSD, database updates over NFS can be an issue as file locking is relatively slow; if the journal is NFS-mounted on these platforms, all operations will be slower. In general (but in particular on Linux and FreeBSD) we recommend that the Perforce database, depot, and journal files be stored on disks local to the machine running the Perforce server process.

These issues affect only the Perforce server process (`p4d`). Perforce clients (such as the `p4` command-line client) have always been able to work with client workspaces on NFS-mounted drives (for instance, workspaces in users' home directories).

Disk space allocation

Perforce disk space usage is a function of three variables:

- Number and size of client workspaces
- Size of server database
- Size of server's archive of all versioned files

All three variables depend on the nature of your data and how heavily you use Perforce.

The client file space required is the size of the files that your users will need in their client workspaces at any one time.

The server's database size can be calculated with a fair level of accuracy; as a rough estimate, it requires 0.5KB per user per file. (For instance, a system with 10,000 files and 50 users will require 250M of disk space for the database). The database can be expected to grow over time as histories of the individual files grow.

The size of the server's archive of versioned files depends on the sizes of the original files stored and grows as revisions are added. For most sites, allocate space equivalent to at least three times the aggregate size of the original files.

If you anticipate your database growing into the gigabyte range, you should ensure that your platform has adequate support for large filesystems. See "Allocate disk space for anticipated growth" on page 20.

The `db.have` file holds label contents and the list of files opened in client workspaces, and tends to grow the faster than other files in the database. If you are experiencing issues related to the size of your `db.have` file and are unable to quickly switch to a server with adequate support for large files, deleting unused clients and labels and reducing the scope of client views can help alleviate the problem.

Network

Perforce can run over any TCP/IP network. Although we have not yet seen network limitations, the more bandwidth the better. Presumably FDDI would be better than 10Mb/s Ethernet, but some users have reported that using a T1 (1.5 Mb/s) provides response times comparable to using Perforce locally. Perforce employees work successfully over ISDN (64 Kb/s) lines.

Perforce uses a TCP connection for each client interaction with the server. The server's port address is defined by `P4PORT`, but the TCP/IP implementation picks a client port number. After the command completes and the connection is closed, the port is left in a state called `TIME_WAIT` for two minutes. While the port number ranges from `1025` to `32767`, generally only a few hundred or thousand can be in use simultaneously. It is therefore possible to occupy all available ports by invoking a Perforce client command many times in rapid succession, such as with a script.

Before release 99.2, both the server and client side of the connection remained in `TIME_WAIT`, which meant that a script running on one user's machine could deprive other users of service by tying up all available ports on the server side. As of Release 99.2, only the client side goes into `TIME_WAIT`, leaving the Perforce server free to handle other clients.

CPU

Perforce is based on a client/server architecture. Both the client and server are lightweight in terms of CPU resource consumption. By way of example, a server supporting 80 users on a low-end (140 MHz) SPARC Ultra server can use as little as 7 CPU-minutes per day, or about 0.5% of available processing power. Weighting this for peak use and headroom, such a server could support upwards of 800 users.

In general, CPU power is not a major consideration when determining the platform on which to install a Perforce server.

Diagnosing Slow Response Times

Perforce is normally a light user of network resources. While it is possible that an extremely large user operation could cause the Perforce server to respond slowly, consistently slow responses to `p4` commands are usually caused by network problems. Any of the following may cause slow response times:

1. misconfigured domain name system (DNS)
2. misconfigured Windows networking
3. difficulty accessing the `p4` executable on a networked file system

A good initial test is to run `p4 info`. If this does not respond immediately, then there is a network problem. Although solving network problems is beyond the scope of this manual, here are some suggestions for troubleshooting them:

Hostname vs. IP address

On a client machine, try setting `P4PORT` to the server's IP address instead of its hostname. For example, instead of using

```
P4PORT=host.domain:1666
```

try using:

```
P4PORT=1.2.3.4:1666
```

with your site-specific IP address and port number.

On most systems, you can determine the IP address of a host by invoking:

```
ping hostname
```

If `p4 info` responds immediately when you use the IP address, but not when you use the hostname, the problem is likely related to DNS.

Try p4 info vs. P4Win

If you are using P4Win on a Windows client, you can compare the response of P4Win "Show Connection Info" (**Help -> Show Connection Info**) with the response from the command-line `p4 info`.

If the former is fast and the latter is slow, you have a DNS-related problem. (When the Perforce server receives a `p4 info` request, it does a reverse name lookup in order to send back the client and server hostnames along with other configuration information. When it

receives a P4Win"Show Connection Info" request, however, it simply returns the IP addresses.)

Note This test is only valid for Release 99.1 and newer servers. In releases prior to 99.1, the server always did a reverse name lookup, whether the request was coming from `p4 info` or P4win

Windows wildcards

In some cases, `p4` commands using unquoted filepatterns with a combination of depot syntax and wildcards, such as:

```
p4 files //depot/*
```

can result in a delayed response on Windows. You can prevent the delay by putting double quotes around the file pattern, like so:

```
p4 files "//depot/*"
```

The cause of the problem is the `p4` command's use of a Windows function to expand wildcards. When quotes are not used, the function interprets `//depot` as a networked computer path and spends time in a futile search for a machine on the network named `depot`.

DNS lookups and the hosts file

On Windows, the `%SystemRoot%\system32\drivers\etc\hosts` file can be used to hardcode IP address-hostname pairs. You may be able to work around DNS problems by adding entries to this file.

The corresponding UNIX file is `/etc/hosts`.

Location of the "p4" executable

If none of the above diagnostic steps explains the sluggish response time, it's possible that the `p4` executable itself is on a networked file system which is performing very poorly. To check this, try running:

```
p4 -V
```

This merely prints out the version information, without attempting any network access. If you get a slow response, network access to the `p4` executable itself may be the problem. Copying or downloading a copy of `p4` onto a local filesystem should improve response times.

Preventing Server Swamp

Generally, Perforce's performance depends on the number of files a user tries to manipulate in a single command invocation, not the size of the depot. That is, syncing a client view of 30 files from a 3,000,000-file depot should not be much slower than syncing a client view of 30 files from a 30-file depot.

The number of files affected by a single command is largely determined by:

- `p4` command line arguments (or selected folders in the case of GUI operations).

  Without arguments, most commands will operate on, or at least refer to, all files in the view.

- Client views, branch views, label views, and protections.

  Because commands without arguments operate on all files in the view, it follows that the use of unrestricted views and unlimited protections can result in commands operating on all files in the depot.

When the server answers a request, it locks down the database for the duration of the computation phase. For normal operations, this is a successful strategy, as it can "get in and out" quickly enough to avoid a backlog of requests. Abnormally large requests, however, can take seconds, sometimes even minutes. If frustrated users hit CTRL-C and retry, the problem gets even worse; the server consumes more memory and responds even more slowly.

At sites with very large depots, unrestricted views and unqualified commands will make a Perforce server work much harder than it needs to. Users and administrators can ease load on their servers by:

- Using "tight" views
- Assigning protections
- Limiting `maxresults`
- Writing efficient scripts

Using tight views

The following "loose" view is trivial to set up but could invite trouble on a very large depot:

```
//depot/...          //workspace/...
```

In the loose view, the entire depot was mapped into the client workspace; for most users, this can be "tightened" considerably. The following view, for example, is restricted to specific areas of the depot:

```
//depot/main/srv/devA/...          //workspace/main/srv/devA/...
//depot/main/drv/lport/...         //workspace/main/dvr/lport/...
//depot/rel2.0/srv/devA/bin/...    //workspace/rel2.0/srv/devA/bin/...
//depot/qa/s6test/dvr/...          //workspace/qa/s6test/dvr/...
```

Client views, in particular, but also branch views and label views, should also be set up to give users just enough scope to do the work they need to do.

Client, branch, and label views are set by the Perforce superuser or by individual users with the `p4 client`, `p4 branch`, and `p4 label` commands respectively.

Two of the techniques for script optimization (described in "Using branch views" on page 101 and "The temporary client trick" on page 102) rely on similar techniques. By limiting the size of the view available to a command, fewer commands need to be run, and when run, the commands require fewer resources.

Assigning protections

Protections (see "Administering Perforce: Protections" on page 61) are actually another type of Perforce view. Protections are set with the `p4 protect` command and control which depot files can be affected by commands run by users.

Unlike client, branch, and label views, however, the views used by protections can be set only by Perforce superusers. (Protections also control read and write permission to depot files, but the permission levels themselves have no impact on server performance.) By assigning protections in Perforce, a Perforce superuser can effectively limit the size of a user's view, even if the user is using "loose" client specifications.

Protections can be assigned to either users or groups. For example:

```
write     user      sam           *     //depot/admin/...
write     group     rocketdev     *     //depot/rocket/main/...
write     group     rocketrel2    *     //depot/rocket/rel2.0/...
```

Perforce groups are created by superusers with the `p4 group` command. Not only do they make it easier to assign protections, but they provide useful fail-safe mechanisms in the form of `maxresults` and `maxscanrows`, described in the next section.

Limiting database queries

Each Perforce group has an associated *maxresults* and *maxscanrows* value. The default for each is "`unlimited`", but a superuser can use `p4 group` to limit it for any given group.

Users in such groups are unable to run any commands which affect more database rows than the group's `maxresults` limit. (For most commands, the number of database rows affected is roughly equal to the number of files affected.)

Like `maxresults`, `maxscanrows` prevents certain user commands from placing excessive demands on the server. (For most commands, the number of rows that could be scanned is roughly equal to the number of files affected, multiplied by the average number of revisions per file in the depot.)

To set these limits, fill in the `Maxresults:` or `Maxscanrows:` field in the `p4 group` form. If a user is listed in multiple groups, the *highest* of the maxresults (or maxscanrows) *limits* (but *not* including the default "`unlimited`" setting) for those groups is taken as the user's `maxresults` (or `maxscanrows`) value.

Example: *Effect of setting* `maxresults` *and* `maxscanrows`*:*

As an administrator, you wish members of the group `rocketdev` *to be limited to operations of 20,000 files or less, and to scan no more than 100,000 revisions:*

```
Group:        rocketdev
Maxresults:   20000
Maxscanrows:  100000
Users:
        bill
        ruth
        sandy
```

Suppose that Ruth has an unrestricted ("loose") client view. When she types:

```
p4 sync
```

her `sync` *command is rejected if the depot contains more than 20,000 files. She can work around this limitation either by restricting her client view, or, if she needs all of the files in the view, by syncing smaller sets of files at a time, like so:*

```
p4 sync //depot/projA/...
p4 sync //depot/projB/...
```

Either way, she'll get her files, but without tying up the server to process a single extremely large command.

If Ruth tries a command that scans every revision of every file, such as:

```
p4 filelog //depot/projA/...
```

and there are less than 20,000 files, but more than 100,000 revisions (perhaps the projA directory contains 8000 files, each of which has 20 revisions), the `maxresults` *limit will not apply, but the* `maxscanrows` *limit will.*

To remove any limits on the number of result lines processed (or database rows scanned) for a particular group, set the `Maxresults:` or `Maxscanrows:` value for that group to `unlimited`.

As these limitations can make life difficult for your users, do not use them unless you find that certain operations are slowing down your server. The `Maxresults:` value should never be less than 10,000, since certain operations performed by P4Win, the Perforce Windows client, may require a `Maxresults:` value of between 5,000 and 8,000. Similarly, `Maxscanrows` should rarely need to be set below 50,000.

For more information, including a comparison of Perforce commands and the number of files they affect, type:

```
p4 help maxresults
p4 help maxscanrows
```

from the command line.

Maxresults and maxscanrows for users in multiple groups

As mentioned earlier, if a user is listed in multiple groups, the highest maxresults limit of all the groups a user belongs to is the limit that affects the user. The default value of "`unlimited`" is *not* a limit; if a user is in a group where maxresults is set to "`unlimited`", he or she is still limited by the highest maxresults (or maxscanrows) limit of the other groups of which he or she is a member. A user's commands are truly unlimited only when the user belongs to no groups, or when all of the groups of which the user is a member have their maxresults set to "unlimited"

A side effect of this is that you can't create a group that assigns "`unlimited`" maxresults values to superusers, because if any of the users in such a group were to belong to another group, the "`unlimited`" limit from the superuser group would also apply to them. You can get around this by assigning a very high maxresults limit to your superusers group.

For example:

```
Group:        superusers
Maxresults:   10000000
Maxscanrows:  100000000
```

(The largest possible `maxresults` or `maxscanrows` limit is platform-dependent; on most platforms, this is a 32-bit integer.)

Scripting efficiently

The Perforce command-line interface, `p4`, allows you to do anything in a script that you can do interactively. The Perforce server can process commands far faster than users can issue them, so in an all-interactive environment, response time is excellent. However, `p4` commands issued by scripts -- triggers, review daemons, or command wrappers, for example -- can cause performance problems if you haven't paid attention to their efficiency. This is not because `p4` commands are inherently inefficient, but because the way one invokes `p4` as an interactive user isn't necessarily suitable for repeated iterations.

This section points out some common efficiency problems and solutions.

Iterating through files

Each Perforce command issued causes a connection thread to be created and a `p4d` subprocess to be started. Reducing the number of Perforce commands your script runs is the first step to making it more efficient.

To this end, scripts should never iterate through files running Perforce commands when they can accomplish the same thing by running one Perforce command on a list of files and iterating through the command results.

For example, try an approach like this:

```
for i in `p4 diff2 path1/... path2/...`
do
    [process diff output]
done
```

Instead of this:

```
for i in `p4 files path1/...`
do
    p4 diff2 path1/$i path2/$i
    [process diff output]
done
```

Using list input files

Any Perforce command that accepts a list of files as a command line argument can also read the same argument list from a file. Scripts can make use of the list input file feature by building up a list of files first, then passing the list file to `p4 -x`.

For example, if your script currently does something like:

```
for components in foo bar ola
do
    p4 edit ${component}.h
done
```

a more efficient alternative would be:

```
for components in foo bar ola
do
    echo ${component}.h >> LISTFILE
done
p4 -x LISTFILE edit
```

The `-x` flag instructs `p4` to read arguments, one per line, from the named file. If the file is specified as "`-`" (a dash), the standard input is read.

Using branch views

Branch views can be used with `p4 integrate` or `p4 diff2` to reduce the number of Perforce command invocations. For example, if you have a script that runs:

```
p4 diff2 pathA/src/...   pathB/src/...
p4 diff2 pathA/tests/... pathB/tests/...
p4 diff2 pathA/doc/...   pathB/doc/...
```

you can make it more efficient by creating a branch view that looks like this:

```
Branch:        pathA-pathB
View:
        pathA/src/...      pathB/src/...
        pathA/tests/...    pathB/tests/...
        pathA/doc/...      pathB/doc/...
```

and replacing the three commands with one:

```
p4 diff2 -b pathA-pathB
```

Limiting label references

Repeated references to large labels can be particularly costly. Commands that refer to files using labels as revisions will scan the whole label once for each file argument. To keep from hogging the Perforce server, your script should get the labeled files from the server, then scan the output for the files it needs.

For example, this:

```
p4 files path/...@label | egrep "path/foo.h|path/bar.h|path/ola.c"
```

will impose a lighter load on the Perforce server than either this:

```
p4 files path/foo.h@label path/bar.h@label path/ola.h@label
```

or this:

```
p4 files path/foo.h@label
p4 files path/bar.h@label
p4 files path/ola.h@label
```

The "temporary client" trick described below may also reduce the number of times you have to refer to files by label.

The temporary client trick

Most Perforce commands can process all the files in the current client view with a single command line argument. By making use of a temporary client view that contains the files on which you want to work, you may be able to reduce the number of commands you have to run, and/or to reduce the number of file arguments you need to give each command.

For instance, suppose your script runs these commands:

```
p4 sync pathA/src/...@label
p4 sync pathB/tests/...@label
p4 sync pathC/doc/...@label
```

You can combine the command invocations and reduce the three label scans to one by using a client spec that looks like:

```
Client:        XY-temp
View:
        pathA/src/...      //XY-temp/pathA/src/...
        pathB/tests/...    //XY-temp/pathB/tests/...
        pathC/doc/...      //XY-temp/pathC/doc/...
```

and running:

```
p4 -c XY-temp sync @label
```

Checkpoints for Database Tree Rebalancing

Perforce's internal database stores its data in structures called Bayer trees, more commonly referred to as B-trees. While B-trees are a very common way to structure data for rapid access, over time the process of adding and deleting elements to and from the trees can eventually lead to imbalances in the data structure.

Eventually, the tree may become sufficiently unbalanced that performance is negatively affected. The Perforce checkpoint and restore processes re-create the trees in a balanced manner, and consequently, you may see some increase in server performance following a restoration from a checkpoint.

Rebalancing the trees is normally only useful if the database files have become more than about 10 times the size of the checkpoint. Given the length of time required for the trees to become unbalanced during normal Perforce use, we expect that the majority of sites will never need to restore the database from a checkpoint (that is, rebalance the trees) for performance reasons.

Chapter 8 Perforce and Windows

This chapter describes certain information of specific interest to administrators who set up and maintain Perforce servers on Windows.

Note Unless otherwise specified, the material presented here applies equally to Windows NT and Windows 2000.

Using the Perforce installer

The Perforce installer program, `perforce.exe`, gives you the option to install either as a user (the Perforce client), a typical administrator (Perforce installed as a Windows service), a custom administrator (Perforce installed as a service with additional customization options), or to uninstall Perforce from your system.

If you have Administrator privileges, it is usually best to install Perforce as a service. If you don't, install it as a server.

Under Windows 2000 or higher, you need Administrator privileges to install Perforce as a service, and Power User privileges to install Perforce as a server.

Upgrade notes

The Perforce installer also automatically upgrades all types of Perforce servers (or services), even versions prior to 97.3. The upgrade process is extremely conservative; if anything fails at any step in the upgrade process, the installer will stop the upgrade, and you will still be able to use your old server (or service).

Installation options

When you invoke the installer, it presents an initial screen that lists the revisions of the Perforce software you're about to install. You are offered the choice between:

- a user install,
- a typical Administrator install,
- a customized Administrator install, or
- uninstalling Perforce.

User install

The "user install" installs only the Perforce command-line client (`p4.exe`), Windows client (P4Win), and (optionally) the third-party SCM plug-in. If you are running on Windows 95 or Windows 98, this will be the only installation option available to you. Under Windows 2000 or higher, this option requires Power User privileges.

You are prompted to specify the location of the client executables, the port (`P4PORT`) on which the client should attempt to contact the Perforce server, the default editor, and the default username.

When specifying the port for the client to use, don't forget to include the hostname in the form *`hostname:port`*. See "Telling Perforce client programs which port to talk to" on page 13 for details on `P4PORT`.

If the installer detects older versions of Perforce client or server software on the machine, you are given the option to rename the old executables to prevent `PATH`-dependent conflicts.

Administrator typical

The "typical administrator install" allows the installation of both client and server software for Perforce. This option requires administrator privileges.

You are prompted to specify the directory for the client and server executables, the port on the local machine where the Perforce server or service will listen to client requests (`P4PORT`), the default editor, and the default username.

The installer selects default locations for the `P4LOG` error log file and the `journal` file. If an earlier version of Perforce was installed on the machine, these locations are based on those already in use.

If you have Administrator privileges, the installer installs Perforce and configures it to run as an auto-starting service. The service is set up and started after the installation is complete, and automatically restarts whenever the machine is rebooted. If you do not have Administrator privileges, a shortcut to run Perforce as a server is placed into your Start menu.

If the installer detects older versions of Perforce client or server software on the machine, you are given the option to rename the old executables to prevent `PATH`-dependent conflicts.

Administrator custom

The "custom administrator install" allows the installation of both client and server software for Perforce. This option requires administrator privileges.

As with the typical administrator install, you are prompted to specify the location of client and server executables, the port on the local machine where the Perforce server or service will listen to client requests, the default editor, and the default username.

Unlike the typical administrator install, you are allowed to optionally specify separate directories for the client and server executables, as well as server root, server port, and whether to set up Perforce as an auto-starting (or non-auto-starting) service or server process. The locations of any existing `P4LOG` file and `journal` file are displayed for reference, and may be changed later using `p4 set`.

If you try to install a Perforce service while another Perforce server is running, you will see the error message:

```
Setup has determined that a Perforce Server could be running. Please
shut down all Perforce Servers before continuing the installation.
```

Failure to shut down the running Perforce server(s) will result in conflicts between the newly installed service and the existing server.

As with the other installation options, if the installer detects older versions of Perforce client or server software on the machine, you are given the option to rename the old executables to prevent `PATH`-dependent conflicts.

Uninstalling Perforce

Should you wish to remove Perforce from a Windows machine, run `perforce.exe` and select the **Uninstall** option. This option requires administrator privileges.

The uninstall procedure removes everything *except* your server data; the Perforce server, service, and client executables, registry keys, and service entries are all deleted. The database and depot files in your server root, however, are always preserved.

Windows services vs. Windows servers

To run any task as a Windows *server*, a user account must be logged in, as shortcuts in a user's `Startup` folder cannot be run until that user logs in. A Windows *service*, on the other hand, is invoked automatically at boot time, and runs regardless of whether or not a user is logged in to the machine.

Throughout most of the documentation set, the terms "Perforce server" or "`p4d`" are used to refer to "the process at the back end that manages the database and responds to requests from Perforce clients". Under Windows, this can lead to ambiguity; the back-end process can run as either a service (`p4s.exe`, which runs as a thread) or as a server (`p4d.exe`, which runs as a regular process). From a Windows administrator's point of view, these are important distinctions. Consequently, the terminology used in this chapter uses the more precise definitions.

The Perforce service (`p4s.exe`) and the Perforce server (`p4d.exe`) executables are copies of each other; they are identical apart from their filenames. When run, they use the first three characters of the name with which they were invoked (that is, either `p4s` or `p4d`) to determine their behavior. For example, invoking copies named `p4smyserver.exe` or `p4dmyservice.exe` will invoke a service and a server, respectively.

Starting and stopping the Perforce service

If Perforce was installed as a service, a user with Administrator privileges can start and stop it using the **Services** applet in the **Control Panel**.

If you are running at Release 99.2 or above, you can also use the command:

```
p4 admin stop
```

to stop the Perforce service.

Starting and stopping the Perforce server

If Perforce was installed as a server, there should be a "Perforce Server" shortcut in your **Start** menu. To start the server, double-click on the shortcut. To stop the server, right-click on the "Perforce Server" button in the taskbar and select "**Close**".

You can also start the Perforce server manually from an MS-DOS window. The server executable, `p4d.exe`, is normally found in your `P4ROOT` directory. To start the server, first make sure your current `P4ROOT`, `P4PORT`, `P4LOG`, and `P4JOURNAL` settings are correct, then run:`%P4ROOT%\p4d`

If you want to start a server using settings different than those set by `P4ROOT`, `P4PORT`, `P4LOG`, or `P4JOURNAL`, you can use `p4d` command line flags. For example:

```
c:\test\p4d -r c:\test -p 1999 -L c:\test\log -J c:\test\journal
```

will start a Perforce server process with a root directory of `c:\test`, listening to port `1999`, logging errors to `c:\test\log`, and with a journal file of `c:\test\journal`.

Note that `p4d` command line flags are case sensitive.

If you are running at Release 99.2 or above, use the following command:

```
p4 admin stop
```

to stop the Perforce server.

> **Note** If you are running Release 99.1 or earlier, type Ctrl-C in the MS-DOS window, or simply **Close** the MS-DOS window.
>
> Although this works in all versions of Perforce, this method of shutting down the server is not necessarily "clean", and with the availability of the `p4 admin stop` command in 99.2, is no longer recommended.

Installing the Perforce service on a network drive

By default, the Perforce service runs under the local `System` account. Because the `System` account has no network access, a real userid and password are required in order to make the Perforce service work if the metadata and depot files are stored on a network drive.

If you are installing your server root on a network drive, the Perforce installer (`perforce.exe`) requests a valid combination of userid and password at the time of installation. This user must have administrator privileges. (The service will also run as this user, rather than `System`)

While the Perforce NT service will run reliably using a network drive as the server root, there is still a marked performance penalty due to of increased network traffic and slower file access. For this reason, we recommend that the depot files and Perforce database be on a drive local to the machine on which the Perforce service is running.

Multiple Perforce services under Windows

By default, the Perforce installer for Windows installs a single Perforce server as a single service. If you wish to host more than one Perforce installation on the same machine (for instance, one for production and one for testing), then the additional Perforce servers must be started manually.

You can, however, install additional services to start up additional Perforce servers (that is, running as Windows services) automatically.

> **Note** You must download Perforce 99.1/10994 or a later release in order to use this procedure.

If your intent is to set up multiple services to increase the number of users you support without purchasing more user licenses, you are violating the terms of your Perforce License Agreement.

Before you begin, you should read and understand “Windows configuration parameter precedence” on page 108. Once you understand the precedence of environment variables in determining Perforce configuration, you’ll find setting up multiple Perforce services to be straightforward. To add a new service, you need to:

1. Create a new directory for the Perforce service.
2. Copy the server executable, service executable, and your license file into this directory.
3. Create the new Perforce service using the `svcinst.exe` utility, as described in the example below.
4. Set up the environment variables and start the new service.

We recommend that you install your first Perforce service using the Perforce installer. This first service will be called "`Perforce`" and its server root directory will contain a few files which are required for the other Perforce services.

Example: *Adding a second Perforce service.*

You wish to create a second Perforce service with a root in `C:\p4root2` *and a service name of "*`Perforce2`*". You're running Release 99.1/10994 or greater, and the* `svcinst` *executable is in the server root of his first Perforce* NT *service, which is in* `C:\perforce`*.*

Verify that your `p4d.exe` *executable is at Release 99.1/10994 or greater:*

```
p4d -V
```

(We'll assume the revision level is correct; if not, you'd have to download a newer version from `http://www.perforce.com` *and upgrade the server before continuing.)*

Create a `P4ROOT` *directory for the new service:*

```
mkdir c:\p4root2
```

Copy the server executables - both `p4d.exe` *(the server) and* `p4s.exe` *(the service) - and your license file into the new directory:*

```
copy c:\perforce\p4d.exe c:\p4root2
copy c:\perforce\p4d.exe c:\p4root2\p4s.exe
copy c:\perforce\license c:\p4root2\license
```

Use `svcinst.exe`*, (the service installer) to create the "*`Perforce2`*" service:*

```
svcinst create -n Perforce2 -e c:\p4root2\p4s.exe -a
```

After the "`Perforce2`*" service has been created, set the service parameters for the "*`Perforce2`*" service:*

```
p4 set -S Perforce2 P4ROOT=c:\p4root2
p4 set -S Perforce2 P4PORT=1667
p4 set -S Perforce2 P4LOG=log2
p4 set -S Perforce2 P4JOURNAL=journal2
```

Finally, use the service installer to start the "`Perforce2`*" service:*

```
svcinst start -n Perforce2.
```

The second service is now running, and both services will start automatically the next time you reboot.

Windows configuration parameter precedence

Under Windows, Perforce configuration parameters may be set in many different ways. When a Perforce client program (such as `p4` or P4Win), or a Perforce server program (`p4d`) starts up, it reads its configuration parameters according to the following precedence:

1. The program's command line flags have the highest precedence.

2. The `P4CONFIG` file, if `P4CONFIG` is set.
3. User environment variables.
4. System environment variables.
5. The Perforce user registry (set by `p4 set`).
6. The Perforce system registry (set by `p4 set -s`).

As of Release 99.1 / 10994, when a Perforce service (`p4s`) starts up, it reads its configuration parameters from the environment according to the following precedence:

1. Windows service parameters (set by `p4 set -S` *`servicename`*) have the highest precedence.
2. System environment variables.
3. The Perforce system registry (set by `p4 set -s`).

User environment variables can be set with any of the following:

- The MS-DOS `set` command.
- The `AUTOEXEC.BAT` file.
- **Control Panel**>**System**>**Environment**>**User Variables**

System environment variables can be set with:

- **Control Panel**>**System**>**Environment**>**System Variables**

Resolving Windows-related instabilities

There are many large sites running Perforce on Windows without incident. There are also sites in which Perforce service or server installation appears to be unstable; the server dies mysteriously, the service can't be started, and in extreme cases the system crashes. In most of these cases, this is an indication of recent changes to the machine or a corrupted operating system.

While not all Perforce failures are caused by OS-level problems, a number of symptoms may indicate the OS is at fault. Examples include: the system crashing, the Perforce server exiting without any error in its log and without Windows indicating that the server crashed, or the Perforce NT service not starting properly.

Many of these problems may be resolved by installing Service Packs. A machine running NT 4.0 with Service Pack 6 and a plain VGA video driver is very stable.

In particular, Perforce support personnel have also found that many sites have been able to reduce or eliminate OS-level instability problems by re-installing NT Service Pack 6;

consequently, we recommend that SP6 be installed on any NT system running the Perforce server or service.

In some cases, installing third-party software *after* installing a Service Pack can overwrite critical files installed by the service pack; reinstalling your most-recently installed service pack will often correct these problems. If you've installed another application after your last service pack, and server stability appears affected since the installation, consider reinstalling the service pack.

As a last resort, it may pay to install Perforce on another system to see if the same failures occur, or even to reinstall the OS and Perforce on the faulty system.

Users having trouble with P4EDITOR or P4DIFF

Your Windows users may experience difficulties using the command line version of the Perforce client (`p4.exe`) if they use the `P4EDITOR` or `P4DIFF` environment variables.

The reason for this is that Perforce clients sometimes use the DOS shell (`cmd.exe`) to start programs such as user-specified editors or diff utilities. Unfortunately, the DOS shell knows that when a Windows command is run (such as a GUI-based editor like `notepad.exe`), it doesn't have to wait for the command to complete before terminating. When this happens, the Perforce client then mistakenly believes that the command has finished, and attempts to continue processing, often deleting the temporary files that the editor or diff utility had been using, leading to errors about temporary files not being found, or other strange behavior.

You can get around this problem in two ways:

- Unset the environment variable `SHELL`. Perforce clients under Windows only use `cmd.exe` when `SHELL` is set, otherwise they call `spawn()` and wait for the Windows programs to complete.
- Set the `P4EDITOR` or `P4DIFF` variable to the name of a DOS batch file, whose contents are the command:

  ```
  start /wait program %1 %2
  ```

  where *`program`* is the name of the editor or diff utility you wish to invoke. The `/wait` flag instructs the system to wait for the editor or diff utility to terminate, and the Perforce client will then behave properly.

  Some Windows editors (most notably, Wordpad) do not behave properly, even when instructed to wait. There is presently no workaround for such programs.

Appendix A Perforce Server (p4d) Reference

Synopsis

Invoke the Perforce server or perform checkpoint/journaling (system administration) tasks.

Syntax

```
p4d [ options ]
p4d.exe [ options ]
p4s.exe [ options ]
p4d -j [ -z ] [ args ... ]
```

Description

The first three forms of the command invoke the Perforce background process ("Perforce server"). The fourth form of the command is used for system administration tasks.

On UNIX and MacOS X, the executable is `p4d`.

On Windows, the executable is `p4d.exe` (running as a server) or `p4s.exe` (running as a service).

Exit Status

After successful startup, `p4d` does not normally exit. It merely outputs the startup message

```
Perforce server starting...
```

and runs in the background.

On failed startup, `p4d` returns a nonzero error code.

Also, if invoked with any of the `-j` checkpointing and/or journalling flags, `p4d` exits with a nonzero error code if any error occurs.

Options

Flag	Meaning
`-d`	Run as a daemon (in the background)
`-f`	Run as a single-threaded (non-forking) process
`-i`	Run from `inetd` on UNIX
`-q`	Run quietly (no startup messages)

Flag	Meaning
`-s`	Run `p4d.exe` as an NT service (equivalent to running `p4s.exe`)
`-xu`	Run database upgrades and exit.
`-xi`	Irreversibly reconfigure the Perforce server (and its metadata) to operate in internationalized mode. Do not use this flag unless you know you require internationalized mode. See the *Release Notes* for details.
`-jc [ prefix ]`	Journal-create; checkpoint and save / truncate journal.
`-jd [ file ]`	Journal-checkpoint; create checkpoint without saving / truncating journal.
`-jj [ prefix ]`	Journal-only; save and truncate journal without checkpointing.
`-jr file`	Journal-restore; restore metadata from a checkpoint and / or journal file.
`-z`	Compress (in `gzip` format) checkpoints and journals.
`-h, -?`	Print help message.
`-V`	Print server version.
`-J journal`	Specify a journal file. Overrides `P4JOURNAL` setting. Default is `journal`.
`-L log`	Specify a log file. Overrides `P4LOG` setting. Default is stderr.
`-p port`	Specify a port to listen to. Overrides `P4PORT`. Default `1666`.
`-r root`	Specify the server root directory. Overrides `P4ROOT`. Default is current working directory.
`-v debuglevel`	Set server trace flags. Overrides value `P4DEBUG` setting. Default is null.

Usage Notes

- On all systems, journaling is enabled by default. If `P4JOURNAL` is unset when a server starts, the default location for the journal is `$P4ROOT/journal`. If you wish to manually disable journaling, you must explicitly set `P4JOURNAL` to `off`.
- Take checkpoints and truncate the journal often, preferably as part of your nightly backup process.
- Checkpointing and journaling preserve only your Perforce metadata (data *about* your stored files). The stored files themselves (the files containing your source code) reside under `P4ROOT` and must be also be backed up as part of your regular backup procedure.

- If your users are using triggers, don't use the `-f` (non-forking mode) flag; the Perforce server needs to be able to spawn copies of itself ("fork") in order to run trigger scripts.
- After a hardware failure, the flags required for restoring your metadata from your checkpoint and journal files may vary, depending on whether or not data was corrupted.
- Because restorations from backups involving loss of files under `P4ROOT` often require the journal file, we strongly recommend that the journal file reside on a separate filesystem from `P4ROOT`. This way, in the event of corruption the filesystem containing `P4ROOT`, the journal is likely to remain accessible.
- The database upgrade flag (`-xu`) is may require considerable disk space. See the *Release Notes* and the section "Important Notes for 2001.1 and later" on page 16 if upgrading to 2001.1 or later from a 2000.2 or earlier server.

Related Commands

To start the server, listening to port `1999`, with journaling enabled and written to `journalfile`.	`p4d -d -p 1999 -J /opt/p4d/journalfile`

To checkpoint a server with a non-default journal file, the `-J` argument (or the environment variable `P4JOURNAL`) must match the journal file specified when the server was started.	Checkpoint with:
	`p4d -jc -J /p4d/jfile`
	or
	`P4JOURNAL=/p4d/jfile ; export P4JOURNAL`
	`p4d -jc -J`

To create a compressed checkpoint from a server with files in directory `P4ROOT`	`p4d -r $P4ROOT -z -jc`
To create a compressed checkpoint with a user-specified prefix of "`ckp`" from a server with files in directory `P4ROOT`	`p4d -r $P4ROOT -z -jc ckp`
To restore metadata from a checkpoint named `checkpoint.3` for a server with root directory `P4ROOT`	`p4d -r $P4ROOT -jr checkpoint.3`
To restore metadata from a compressed checkpoint named `checkpoint.3.gz` for a server with root directory `P4ROOT`	`p4d -r $P4ROOT -z -jr checkpoint.3.gz`

Index