Perforce 2009.2
System Administrator's Guide

December 2009

Table of Contents

Preface

About This Manual

This is the *Perforce 2009.2 System Administrator's Guide*.

This guide is intended for people responsible for installing, configuring, and maintaining Perforce servers. This guide covers tasks typically performed by a "system administrator" (for instance, installing and configuring the software and ensuring uptime and data integrity), as well as tasks performed by a "Perforce administrator", such as setting up Perforce users, configuring Perforce depot access controls, resetting Perforce user passwords, and so on.

Because Perforce requires no special system permissions, a Perforce administrator does not typically 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. To familiarize yourself with the Perforce Command-Line Client, see the *Perforce Command Reference*.

Using Perforce?

If you plan to use Perforce as well as administer a Perforce server, see the *P4 User's Guide* for information on Perforce from a user's perspective.

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

Please give us feedback

We are interested in receiving opinions on this manual 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 are upgrading an existing installation to Release 2005.1 or later, see the notes in "Upgrading a Perforce server" on page 18 before proceeding.

This chapter includes a brief overview of things to consider at installation time, along with some basic security and administration tips. More detailed information on administrative tasks is found in later chapters.

Windows Where the UNIX and Windows versions of Perforce differ, this manual notes the difference. For Windows-specific information, see "Perforce and Windows" on page 145.

Many of the examples in this book are based on the UNIX version of the Perforce server. In most cases, the examples apply equally to both Windows and UNIX installations.

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

Getting Perforce

Perforce requires at least two executables: the server (`p4d`), and at least one Perforce client program (such as `p4` on UNIX, or `p4.exe` on Windows).

The server and client executables are available from the Downloads page on the Perforce web site:

```
http://www.perforce.com/perforce/downloads/index.html
```

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

UNIX installation

Although you can install `p4` and `p4d` 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. You can install Perforce client programs on any machine that has TCP/IP access to the `p4d` host.

To limit access to the Perforce server files, ensure that the `p4d` executable is owned and run by a Perforce user account that has been created for the purpose of running the Perforce server.

To start using Perforce:

1. Download the `p4` and `p4d` files for your platform from the Perforce web site.
2. Make the downloaded `p4` and `p4d` files executable.
3. Create a server root directory to hold the Perforce database and versioned files.
4. Tell the Perforce server what port to listen to by specifying a TCP/IP port to `p4d`.
5. Start the Perforce server (`p4d`).
6. Specify the name or TCP/IP address of the Perforce server machine and the `p4d` port number to the Perforce client programs by setting the `P4PORT` environment variable.

Downloading the files and making them executable

On UNIX (or Mac OS X), you must make the Perforce executables (`p4` and `p4d`) executable. After you download the programs, use the `chmod` command to make them executable, as follows:

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

Creating a Perforce server root directory

The Perforce server stores all user-submitted files and system-generated metadata in files and subdirectories beneath its own root directory. This directory is called the *server root*.

To specify a server root, either set the environment variable `P4ROOT` to point to the server root, or use the `-r` *`root_dir`* flag when invoking `p4d`. Perforce client programs never use the `P4ROOT` directory or environment variable; the `p4d` server is the only process that uses the `P4ROOT` variable.

Because all Perforce files are stored beneath the server root, the contents of the server root grow over time. See "Installation and administration tips" on page 19 for a brief overview of diskspace requirements, and "Disk space allocation" on page 134 for more detail.

A Perforce server requires no privileged access; there is no need to run `p4d` as `root` or any other privileged user. For more information, see "UNIX: Run p4d as a nonprivileged user" on page 23.

The server root can be located anywhere, but the account that runs `p4d` must have `read`, `write`, and `execute` permissions on the server root and all directories beneath it. For security purposes, set the `umask(1)` file-creation-mode mask of the account that runs `p4d` to a value that denies other users access to the server root directory.

Telling Perforce servers which port to listen on

The `p4d` server and Perforce client programs communicate with each other using 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 a host named `perforce`, listening on port `1666`.

If `p4d` is to listen on a different port, either specify that port with the `-p` *`port_num`* flag you start `p4d` (as in, `p4d -p 1818`), or set the port with the `P4PORT` environment or registry variable.

Unlike `P4ROOT`, the environment variable `P4PORT` is used by both the Perforce server and the Perforce client programs, so it must be set on both Perforce server machines and Perforce client workstations.

Telling Perforce client programs which port to connect to

Perforce client programs need to know on what machine the `p4d` server resides and on which TCP/IP port the `p4d` server program is listening. 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 number of the port on which `p4d` is listening.

Examples:

P4PORT	Client program behavior
`dogs:3435`	Perforce client programs connect to the `p4d` server on host `dogs` listening on port `3435`.
`x.com:1818`	Perforce client programs connect to the `p4d` server on host `x.com` listening on port `1818`.

If the Perforce client program is running on the same host as `p4d`, only the `p4d` port number need be provided in `P4PORT`. If `p4d` is running on a host named or aliased `perforce`, and is listening on port `1666`, leave `P4PORT` unset. For example:

P4PORT	Client program behavior
`3435`	Perforce client programs connect to the `p4d` server on the same machine as the client program, listening on port `3435`.
`<not set>`	Perforce client programs connect to the `p4d` server on a host named or aliased `perforce` listening on port `1666`.

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

If your network environment and Perforce Server have been configured to support Zeroconf services, users can set `P4PORT` to the value of the service name.

Starting the Perforce server

After you set p4d's P4PORT and P4ROOT environment variables, start the server by running p4d in the background with the command:

```
p4d &
```

Although the example shown is sufficient to run p4d, you can specify other flags that control such things as error logging, checkpointing, and journaling.

Example: *Starting a Perforce server*

You can override P4PORT by starting p4d with the -p flag, and P4ROOT by starting p4d with the -r flag. Similarly, you can specify a journal file with the -J flag, and an error log file with the -L flag. A startup command that overrides the environment variables might look like this:

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

The -r, -J, and -L flags (and others) are discussed in "Supporting Perforce: Backup and Recovery" on page 25. A complete list of server flags is provided in the "Perforce Server (p4d) Reference" on page 167.

Stopping the Perforce server

To shut down a Perforce server, use the command:

```
p4 admin stop
```

Only a Perforce superuser can use p4 admin stop.

If you are running a release of Perforce from earlier than 99.2, you must find the process ID of the p4d server and kill the process manually from the UNIX shell. Use kill -15 (SIGTERM) instead of kill -9 (SIGKILL), because p4d might leave the database in an inconsistent state if it is in the middle of updating a file when a SIGKILL signal is received.

Windows installation

To install Perforce on Windows, use the Perforce installer (perforce.exe) from the Downloads page of the Perforce web site:

```
http://www.perforce.com/perforce/downloads/index.html
```

Use the Perforce installer program to install or upgrade the Perforce Server, Perforce Proxy, or the Perforce Command-Line Client.

Other Perforce clients on Windows, such as the Perforce Visual Client (P4V), as well as third-party plug-ins, may be downloaded and installed separately.

For more about installing on Windows, see "Using the Perforce installer" on page 145.

Windows services and servers

In this manual, the terms *Perforce server* and `p4d` are used interchangeably to refer to "the process which handles requests from Perforce client programs" unless the distinction between a Windows server process a Windows service process is relevant.

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 configured to run as a Windows service (`p4s.exe`) process that starts at boot time, or as a server (`p4d.exe`) process that you invoke manually from a command 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, the executables 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 of `p4d.exe` named `p4smyservice.exe` or `p4dmyserver.exe` invoke a service and a server, respectively.)

In most cases, it is preferable to install Perforce as a service, not a server. For a more detailed discussion of the distinction between services and servers, see "Windows services vs. Windows servers" on page 146.

Starting and stopping Perforce

If you install Perforce as a service under Windows, the service starts whenever the machine boots. Use the **Services** applet in the **Control Panel** to control the Perforce service's behavior.

If you install Perforce as a server under Windows, invoke `p4d.exe` from a command prompt. The flags for `p4d` under Windows are the same as those used under UNIX.

To stop a Perforce service (or server), use the command:

```
p4 admin stop
```

Only a Perforce superuser can use `p4 admin stop`.

For older revisions of Perforce, shut down services manually by using the **Services** applet in the **Control Panel**. Shut down servers running in command prompt windows by pressing CTRL-C in the window or by clicking the icon to close the command prompt window. Although manually shutting down in this way works with Release 99.2 and earlier versions of Perforce, it is 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, this manual shutdown method is obsolete.

Upgrading a Perforce server

You *must* back up your server (see "Backup procedures" on page 31) as part of any upgrade process.

Warning! Before you upgrade a Perforce server, always read the release notes associated with your upgraded server.

Using old client programs with a new server

Although older Perforce client programs generally work with newer server versions, some features in new server releases require upgrades to Perforce client programs. In general, users with older client programs are able to use features available from the Perforce server at the client program's release level, but are not able to use the new server features offered by subsequent server upgrades.

Licensing and upgrades

To upgrade your current Perforce server to a newer version, your Perforce license file must be current. Expired licenses do not work with upgraded servers.

Upgrading the server

Read the *Release Notes* for complete information on server upgrade procedures. In general, Perforce server upgrades require that you:

1. Stop the Perforce Server (`p4d`).
2. Make a checkpoint and back up your old installation.
3. Replace the Perforce Server executable with the upgraded version of the server.

   On UNIX, replace the old version of `p4d` with the new version downloaded from the Perforce website. On Windows, use the Perforce installer (`perforce.exe`); the installer automatically replaces the server executable.
4. Some upgrades (installations with more than 1000 changelists, or upgrades with significant database changes) may require that you manually upgrade the database by running:

   ```
   p4d -r server_root -J journal_file -xu
   ```

   This command may take considerable time to complete. You must have sufficient disk space to complete the upgrade.
5. Restart the Perforce Server with your site's usual parameters.

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 support.

Licensing information is contained in a file called `license` in the server root directory. The `license` file is a plain text file supplied by Perforce Software. Without the `license` file, the Perforce server limits itself to two users and five client workspaces.

You can update an existing file without stopping the Perforce Server by using the `p4 license` command. See "Adding new licensed users" on page 45 for details.

To view current licensing information, invoke `p4d -V` from the server root directory where the `license` file resides, or by specifying the server root directory either on the command line (`p4d -V -r server_root`) or in the `P4ROOT` environment variable.

If the server is running, you can also use `p4 info` to view your licensing information.

The server version is also displayed when you invoke `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.
- 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 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 certain that, if a disk failure corrupts the drive containing `P4ROOT`, such a failure will *not* affect your journal file. You can then use the journal file 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. After you start a new Perforce server, use:

```
p4 protect
```

as soon as possible to define a Perforce superuser. To learn more about how `p4 protect` works, see "Administering Perforce: Protections" on page 81.

Without passwords, any user is able to impersonate any other Perforce user, either with the `-u` flag or by setting `P4USER` to an existing Perforce user name. Use of Perforce passwords prevents such impersonation. See the *P4 User's Guide* for details.

To set (or reset) a user's password, either 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 is supported in release 99.1 or later; the latter command is supported under all releases from 97.3 onwards.

The security-conscious Perforce superuser also uses `p4 protect` to ensure that no access higher than `list` is granted to nonprivileged users, and to ensure that each user has a Perforce password.

Allocate sufficient disk space for anticipated growth

Because the collection of versioned files grows over time, a good guideline is to allocate sufficient space in your `P4ROOT` directory to hold three times the size of your users' present collection of versioned files, plus an additional 0.5KB per user per file to hold the database files that store the list of depot files, file status, and file revision histories.

For a more detailed example of a disk sizing estimate, see "Disk space allocation" on page 134.

Managing disk space after installation

All of Perforce's versioned files reside in subdirectories beneath the server root, as do the database files, and (by default) the checkpoints and journals. If you are running low on disk space, consider the following approaches to limit disk space usage:

- Configure 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.
- Keep the journal file short by taking checkpoints on a daily basis.
- Compress checkpoints, or use the `-z` option to tell `p4d` to compress checkpoints on the fly.

- 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 to a different drive and then delete the copied checkpoints from the root directory. Moving checkpoints to separate drives is good practice not only in terms of diskspace, but also because old checkpoints are needed when recovering from a hardware failure, 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, you can relocate some or all of the depot directories to other disks by using symbolic links. If you use symbolic links to shift depot files to other volumes, create the links only after you stop the Perforce server.
- If your installation's database files have grown to more than 10 times the size of a checkpoint, you might be able to reduce the size of the files by re-creating them from a checkpoint. See "Checkpoints for database tree rebalancing" on page 144.
- Use the `p4 sizes` command to monitor the amount of disk space currently consumed by your entire installation, or by selected portions of your installation. See "Monitoring disk space usage" on page 135.

Large filesystem support

Early versions of the Perforce server, as well as some operating systems, limit Perforce database files (the `db.*` files in the `P4ROOT` directory that hold your site's metadata) to 2 GB in size. The `db.have` file holds the list of files currently synced to client workspaces, and tends to grow the most quickly.

If you anticipate any of your Perforce database files growing beyond the 2 GB level, install the Perforce server on a platform with support for large files. The following combinations of operating system and Perforce server revision support database files larger than 2 GB:

Operating system	OS version	Perforce server revision
Windows NT, 2000, XP, Vista, 7	All versions, SP6 recommended for NT	98.2/8127 or higher
FreeBSD	All versions	98.2/5713 or higher
Linux x86	Kernels 2.4.0 and higher	2002.2/21749 or higher
HP-UX	HP-UX 11.11 and higher	2001.1/26433 or higher
Solaris	2.6 and higher	98.2/7488 *compiled for 2.6 or higher*
Tru64 UNIX (a.k.a. Digital UNIX, OSF/1)	All versions	98.2/5713 or higher

Operating system	OS version	Perforce server revision
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

UNIX and NFS support

To maximize performance, configure the server root (`P4ROOT`) to reside on a local disk and not an NFS-mounted volume. The Perforce Server's file-locking semantics work with NFS mounts on Solaris 2.5.1 and later; some issues still remain regarding file locking on noncommercial implementations of NFS (for instance, Linux and FreeBSD).

These issues affect only the Perforce Server process (`p4d`). Perforce client programs (such as `p4`, the Perforce Command-Line Client) have always been able to work with client workspaces on NFS-mounted drives, such as client 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 requires an account name and a password. The Perforce service is then configured with the supplied data and run as the specified user instead of `System`. (The account running the service must have `Administrator` privileges on the machine.)

Although Perforce operates reliably with its root directory on a network drive, it does so only at a substantial performance penalty, because all writes to the database are performed over the network. For optimal performance, 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 147.

UNIX: Run p4d as a nonprivileged user

The Perforce server process does not require privileged access. For security reasons, do not run `p4d` as `root` or otherwise grant the owner of the `p4d` process `root`-level privileges.

Create a nonprivileged UNIX user (for example, `perforce`) to manage `p4d` and (optionally) a UNIX group for it (for example, `p4admin`). Use the `umask(1)` command to ensure that the server root (`P4ROOT`) and all files and directories created beneath it are writable only by the UNIX user `perforce`, and (optionally) readable by members of the UNIX group `p4admin`.

Under this configuration, the Perforce server (`p4d`), running as UNIX user `perforce`, can write to files in the server root, but no users are able to read or overwrite its files. To grant access to the files created by `p4d` (that is, the depot files, checkpoints, journals, and so on) to trusted users, you can add the trusted users to the UNIX group `p4admin`.

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

Logging errors

Use the `-L` flag to `p4d` or the environment variable `P4LOG` to specify the Perforce server's error output file. If no error output file is defined, errors are dumped to the `p4d` process' 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` also logs these errors to its error output.

The Perforce server also supports trace flags used for debugging. See "Perforce server trace and tracking flags" on page 60 for details.

Logging file access

If your site requires that user access to files be tracked, use the `-A` flag to `p4d` or the environment variable `P4AUDIT` to activate auditing and specify the Perforce server's audit log file. When auditing is active, every time a user accesses a file, a record is stored in the audit log file. This option can consume considerable disk space on an active server.

See "Auditing user file access" on page 62 for details.

Case sensitivity issues

Whether your Perforce Server is running on Windows or UNIX, if your site is involved in cross-platform development (that is, if you are using Perforce client programs on both Windows and UNIX workstations), your users must be aware of certain details regarding case sensitivity issues.

See "Case sensitivity and multiplatform development" on page 57 for details.

Enable server process monitoring

The Perforce Server tracks information about Perforce-related processes running on your Perforce server machine. Server process monitoring requires minimal system resources, but you must enable process monitoring for `p4 monitor` to work. To enable process monitoring, set the `monitor` counter as follows:

```
p4 counter -f monitor 1
```

After you set or change the `monitor` counter, you must stop and restart the Perforce server for your change to take effect.

See "Monitoring server activity" on page 58 for details.

Tune for performance

Perforce is an efficient consumer of network bandwidth and CPU power. The most important variables that determine server performance are 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 133.

Chapter 2 Supporting Perforce: Backup and Recovery

The Perforce server stores two kinds of data: *versioned files* and *metadata*.

- *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 workspace specifications, branch mappings, 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.

The 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 that constitute the Perforce database, on the other hand, are not guaranteed to be in a state of transactional integrity if archived by a conventional backup program. Restoring the `db.*` files from regular system backups can 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 of updates to the database since the last snapshot was taken.

The checkpoint file is often much smaller than the original database, and it 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 versioned files stored in the depot directories!

You must always back up the depot files (your versioned file tree) 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 part of your regular backup cycle.

Checkpoint files

A *checkpoint* is a file that contains all information necessary to re-create the metadata in the Perforce database. When you create a checkpoint, the Perforce database is locked, enabling 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. To create a checkpoint, invoke the `p4d` program with the `-jc` (journal-create) flag:

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

You can create a checkpoint while the Perforce server (`p4d`) is running. The checkpoint is created in your server root directory (`P4ROOT`).

To make the checkpoint, `p4d` locks the database and then dumps its contents to a file named `checkpoint.n` in the `P4ROOT` directory, where `n` is a sequence number. Before unlocking the database, `p4d` also copies (on UNIX where the journal is uncompressed, renames) the journal file to a file named `journal.n-1` in the `P4ROOT` directory (regardless of the directory in which the current journal is stored), and then truncates the current

journal. This guarantees that the last checkpoint (`checkpoint.n`) combined with the current journal (`journal`) always reflects 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, you can restore the database reflected by `checkpoint.6` 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`.

To specify a prefix or directory location for the checkpoint and journal, use the `-jc` option. For example, you might create a checkpoint with:

```
p4d -jc prefix
```

In this case, your checkpoint and journal files are named `prefix.ckp.n` and `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` are used. You can store checkpoints and journals in the directory of your choice by specifying the directory as part of the prefix. (Rotated journals are stored in the `P4ROOT` directory, regardless of the directory in which the current journal is stored.)

To create a checkpoint without being logged in to the machine running the Perforce server, use the command:

```
p4 admin checkpoint [-z] [prefix]
```

Running `p4 admin checkpoint` is equivalent to `p4d -jc`. You must be a Perforce superuser to use `p4 admin`.

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 checkpoint creation was started. After successful creation, a checkpoint file can be compressed, archived, or moved onto another disk. At that time or shortly thereafter, back up the versioned files stored in the depot subdirectories.

To restore from a backup, *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.

If the checkpoint command itself fails, contact Perforce technical support immediately. Checkpoint failure is usually a symptom of a resource problem (disk space, permissions, and so on) 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 filename is `journal`, and the file 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. To do this, specify the name of the journal file in the environment variable `P4JOURNAL` or use the `-J` *`filename`* flag when starting `p4d`.

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 147), 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`*, 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`). Because 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 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 filename 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 -J /usr/local/perforce/journalfile -jc
Checkpointing to checkpoint.19...
Saving journal to journal.18...
Truncating /usr/local/perforce/journalfile...
```

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

```
$ p4d -r $P4ROOT -jc
Checkpointing to checkpoint.19...
Saving journal to journal.18...
Truncating /usr/local/perforce/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 can be one or more archived binary files in each *`filename`*`,d` directory. If more than one file resides in a *`filename`*`,d` directory, each file in the directory refers to a different revision of the binary file, and is named `1.`*`n`*, where *`n`* is the revision number.

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

Because Perforce uses compression in the depot file tree, do not assume 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 they 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.)

Backing 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, create the checkpoint before backing up the versioned files in the depot directory or directories.

Although 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 MD5 digests and file length metadata to any new files:

   ```
   p4 verify //...
   ```

   You might want to use the `-q` (quiet) option with `p4 verify`. If called with the `-q` option, `p4 verify` produces output only when errors are detected.

   The `p4 verify` command recomputes the MD5 signatures of all of your archived files and compares them with those stored when the files were first stored, and that all files known to Perforce exist in the depot subdirectories.

   By running `p4 verify` before the backup, you ensure that you create and store checksums and file length metadata for any files new to the depot since your last backup, and that this information is stored as part of the backup you're about to make.

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

   Note For large installations, `p4 verify` might take some time to run. Furthermore, the database is locked when `p4 verify` is running, which prevents most other Perforce commands from being used. Administrators of large sites might choose to perform `p4 verify` on a weekly basis, rather than a nightly basis.

   For more about the `p4 verify` command, see "Verifying files by signature" on page 47.

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:

   ```
   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 gigabytes of `db.*` files), creating a checkpoint might take a considerable length of time.

Under such circumstances, you might want 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.

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. (In most cases, you don't actually need to back up the journal, but it is usually good practice to do so.)

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

   Most sites are unaffected 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 restart the server only after the backup process has completed. Doing so will eliminate any 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 can fail.

If your backup software doesn't allow you to exclude the `db.*` files from the backup process, 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 or because of a hardware failure such as a disk crash, the database can be re-created with your stored checkpoint and journal.

There are many ways in which systems can fail. Although this guide cannot address all failure scenarios, 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
- 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 database (`db.*`) files:

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

   There can be no `db.*` files in the `$P4ROOT` directory when you start recovery from a checkpoint. Although the old `db.*` files are never used during recovery, it's good 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 applies 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 your 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 you recover 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.

Files submitted to the depot between the time of the last system backup and the disk crash will not be present in the restored depot.

Note Although "new" files (submitted to the depot but not yet backed up) do not appear in the depot after restoration, it's possible (indeed, highly probable!) that 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 the following Perforce command 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 that differ from the files Perforce believes them to have. After verifying that these files are indeed the files you want to restore, you may want to have one of your users open these files for `edit` and submit the files to the depot in a changelist.

Your system state

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

- 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 might have resided on a separate disk and been unaffected by the crash) were lost, you might 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 which files were 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 that 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 produce output only 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` 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 basic tasks associated with day-to-day Perforce administration and advanced Perforce configuration issues related to cross-platform development issues, migration of Perforce servers from one machine to another, and working with remote and local depots.

Most of the tasks described in this chapter requires that you have Perforce superuser (access level `super`) or administrator (access level `admin`) privileges as defined in the Perforce protections table. For more about controlling Perforce superuser access, and protections in general, see "Administering Perforce: Protections" on page 81.

Release 2004.2 of Perforce introduced a new authentication mechanism and a server-configurable security setting to govern password strength requirements and authentication method policy. For details, see "Authentication methods: passwords and tickets" on page 39 and "Server security levels" on page 41.

Basic Perforce Administration

Tasks commonly performed by Perforce administrators and superusers include:

- User maintenance tasks, including resetting passwords, creating users, disabling the automatic creation of users, and cleaning up files left open by former users
- Administrative operations, including setting the server security level, obliterating files to reclaim disk space, editing submitted changelists, verifying server integrity, defining file types to control Perforce's file type detection mechanism, and the use of the `-f` flag to force operations

Authentication methods: passwords and tickets

Perforce supports two methods of authentication: password-based and ticket-based.

Warning Although ticket-based authentication provides a more secure authentication mechanism than password-based authentication, it does not encrypt network traffic between client workstations and the Perforce server.

If you are accessing Perforce over an insecure network, use a third-party tunneling solution (for example, `ssh` or a VPN) regardless of the authentication method you choose.

How password-based authentication works

Password-based authentication is stateless; once a password is correctly set, access is granted for indefinite time periods. Prior to Release 2004.2, the password-based authentication mechanism did not enforce password strength or existence requirements.

The concept of the server security level, introduced in Release 2004.2, enables administrators to enforce password strength and existence requirements. See "Server security levels" on page 41 for details.

Password based authentication is supported at security levels 0, 1, and 2.

How ticket-based authentication works

Ticket-based authentication is based on time-limited tickets that enable users to connect to Perforce servers. Tickets are stored in the file specified by the `P4TICKETS` environment variable. If this variable is not set, tickets are stored in `%USERPROFILE%\p4tickets.txt` on Windows, and in `$HOME/.p4tickets` on UNIX and other operating systems. Tickets are managed automatically by 2004.2 and later Perforce client programs.

Tickets have a finite lifespan, after which they cease to be valid. By default, tickets are valid for 12 hours (43200 seconds). To set different ticket lifespans for groups of users, edit the `Timeout:` field in the `p4 group` form for each group. The timeout value for a user in multiple groups is the largest timeout value (including `unlimited`, but ignoring `unset`) for all groups of which a user is a member. To create a ticket that does not expire, set the `Timeout:` field to `unlimited`.

Although tickets are not passwords, Perforce servers accept valid tickets wherever users can specify Perforce passwords. This behavior provides the security advantages of ticket-based authentication with the ease of scripting afforded by password authentication. Ticket-based authentication is supported at all server security levels, and is required at security level 3.

Logging in to Perforce

To use ticket-based authentication, get a ticket by logging in with the `p4 login` command:

```
p4 login
```

You are prompted for your password, and a ticket is created for you in your ticket file. You can extend your ticket's lifespan by calling `p4 login` while already logged in. If you run `p4 login` while logged in, your ticket's lifespan is extended by 1/3 of its initial timeout setting, subject to a maximum of your initial timeout setting.

By default, Perforce tickets are valid for your IP address only. If you have a shared home directory that is used on more than one machine, you can log in to Perforce from both machines by using the command:

```
p4 login -a
```

to create a ticket in your home directory that is valid from all IP addresses.

Logging out of Perforce

To log out of Perforce from one machine by removing your ticket, use the command:

```
p4 logout
```

The entry in your ticket file is removed. If you have valid tickets for the same Perforce server, but those tickets exist on other machines, those tickets remain present (and you remain logged in) on those other machines.

If you are logged in to Perforce from more than one machine, you can log out of Perforce from all machines from which you were logged in by using the command:

```
p4 logout -a
```

All of your Perforce tickets are invalidated and you are logged out.

Determining ticket status

To see if your current ticket (that is, for your IP address, user name, and `P4PORT` setting) is still valid, use the command:

```
p4 login -s
```

If your ticket is valid, the length of time for which it will remain valid is displayed.

To display all tickets you currently have, use the command:

```
p4 tickets
```

The contents of your ticket file are displayed.

Server security levels

Perforce superusers can configure server-wide password usage requirements, password strength enforcement, and supported methods of user/server authentication by setting the `security` counter. To change the `security` counter, issue the command:

```
p4 counter -f security seclevel
```

where *`seclevel`* is 0, 1, 2, or 3. After setting the counter, stop and restart the server.

Choosing a server security level

The default security level is 0: passwords are not required, and password strength is not enforced.

To ensure that all users have passwords, use security level 1. Users of old client programs can still enter weak passwords.

To ensure that all users have strong passwords, use security level 2. Old Perforce software continues to work, but users of old Perforce client software must change their password to a strong password by using a Perforce client program at Release 2003.2 or above.

To require that all users have strong passwords, and to require the use of session-based authentication, use security level 3 and current Perforce client software.

Level 0 corresponds to pre-2003.2 server operation. Levels 1 and 2 were designed for support of legacy client software. Level 3 affords the highest degree of security.

The Perforce server security levels and their effects on the behavior of Perforce client programs are defined below.

Security level	Server behavior
`0` (or unset)	Legacy support: passwords are not required. If passwords are used, password strength is not enforced.
	Users with passwords can use either their `P4PASSWD` setting or the `p4 login` command for ticket-based authentication.
	Users of old Perforce client programs are unaffected.
`1`	Strong passwords are required for users of post-2003.2 Perforce client programs, but existing passwords are not reset.
	Pre-2003.2 Perforce client programs can set passwords with `p4 passwd` or in the `p4 user` form, but password strength is not enforced.
	Users with passwords can use either their `P4PASSWD` setting or the `p4 login` command for ticket-based authentication.
`2`	All unverified strength passwords must be changed.
	Users of pre-2003.2 client programs cannot set passwords.
	Users of client programs at release 2003.2 or higher must use `p4 passwd` and enter their passwords at the prompt. Setting passwords with the `p4 user` form or the `p4 passwd -O` *`oldpass`* `-P` *`newpass`* command is prohibited.
	On Windows, passwords are no longer stored in (or read from) the registry. (Storing `P4PASSWD` as an environment variable is supported, but passwords set with `p4 set P4PASSWD` are ignored.)
	Users who have set strong passwords with a 2003.2 or higher Perforce client program can use either their `P4PASSWD` setting for password-based authentication, or the `p4 login` command for ticket-based authentication.
`3`	All password-based authentication is rejected.
	Users must use ticket-based authentication (`p4 login`).
	If you have scripts that rely on passwords, use `p4 login` to create a ticket valid for the user running the script, or use `p4 login -p` to display the value of a ticket that can be passed to Perforce commands as though it were a password (that is, either from the command line, or by setting `P4PASSWD` to the value of the valid ticket).

Password strength

Certain combinations of server security level and Perforce client software releases require users to set "strong" passwords. A password is considered strong if it is at least eight characters long, and at least two of the following are true:

- The password contains uppercase letters.
- The password contains lowercase letters.
- The password contains nonalphabetic characters.

For example, the passwords `a1b2c3d4`, `A1B2C3D4`, `aBcDeFgH` are considered strong.

Resetting user passwords

If you are a Perforce superuser, you can reset a Perforce user's password with:

- Release 99.1 and later:

  ```
  p4 passwd username
  ```

  When prompted, enter a new password for user *username*.

- Pre-99.1 releases:

  ```
  p4 user -f username
  ```

  Enter the password in the `Password:` field of the user specification form.

Creating users

By default, Perforce creates a new user record in its database whenever a command is issued by a user that does not exist. Perforce superusers can also 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 want to create.

The `p4 user` command also has an option (`-i`) to take its input from the standard input instead of the forms editor. To quickly create a large number of users, write a script that reads user data, generates output in the format used by the `p4 user` form, and then pipes each generated form to `p4 user -i -f`.

Preventing automatic creation of users

By default, Perforce creates a new user record in its database whenever a user invokes any client command that can update the depot or its metadata.

To prevent Perforce from automatically creating users, all users must be defined in the protections table. The easiest way to do this is to include all users in a Perforce group, and to configure Perforce to grant access only 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: Limits the rows (unless 'unlimited' or 'unset') any
#             one operation can return to the client.
# MaxScanRows:Limits the rows (unless 'unlimited' or 'unset') any
#             one operation can scan from any one database table.
# MaxLockTime:Limits the time (in milliseconds, unless 'unlimited'
#             or 'unset') any one operation can lock any database
#             table when scanning data...
# Timeout:    Time (in seconds, unless 'unlimited' or 'unset') which
#             determines how long a 'p4 login' session ticket
#             remains valid (default is 12 hours).
# Subgroups:  Other groups automatically included in this group.
# Owners:     Users allowed to change this group without requiring
#             super access permission.
# Users:      The users in the group.  One per line.
Group:  p4users
MaxResults:     unset
MaxScanRows:    unset
MaxLockTime:    unset
Timeout:        43200
Subgroups:
Owners:
Users:
        edk
        lisag
```

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* `//...`*).*

Finally, the superuser 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` *can 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 (or deny!) permission to "all users", all users are now defined within `p4 protect`*, and the server no longer automatically creates 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 81.

Deleting obsolete users

Each user on the system consumes one Perforce license. A Perforce administrator can free up licenses by deleting users with the following command:

```
p4 user -d -f username
```

Before you delete a user, you must first revert (or submit) any files a user has open in a changelist. If you attempt to delete a user with open files, Perforce displays an error message to that effect.

Deleting a user frees a Perforce license but does not automatically update the group and protections tables. Use `p4 group` and `p4 protect` to delete the user from these tables.

Adding new licensed users

Perforce licenses are controlled by a text file called `license`. This file resides in the server root directory.

To add or update a license file, stop the Perforce Server, copy the `license` file into the server root directory, and restart the Perforce Server.

As of Release 2006.2, you can update an existing `license` without shutting down the Perforce Server, use `p4 license -i` to read the new license file from the standard input.

Most new license files obtained from Perforce can be installed with `p4 license`, except for when the server IP address has changed. If the server IP address has changed, or if no `license` file currently exists, you must still stop the Perforce Server, manually copy the license file into place, and restart the Server.

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 administrator can revert the files by deleting the client workspace specification in which the files were opened.

As an example, if the output of `p4 opened` includes:

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

you can delete the `stlouis` client workspace specification with:

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

Deleting a client workspace specification automatically reverts all files opened in that workspace, deletes pending changelists associated with the workspace, and any pending fix records associated with the workspace. Deleting a client workspace specification does *not* affect any files in the workspace actually used by the workspace's owner; 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, which is not always desirable: a user might have created hundreds of unneeded files by means of an inadvertent branch or submit, or perhaps there are directories of old files that are no longer in use. Because `p4 delete` merely marks files as deleted in their head revisions, it cannot be used to free up disk space on the server. This is where `p4 obliterate` can be useful.

Perforce administrators 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 enable your site to maintain a history of what operations were performed on which files. The `p4 obliterate` command defeats this purpose; as such, it is intended only to be used to remove files that never belonged in the depot in the first place, and not as part of a normal software development process.

Note also that `p4 obliterate` is computationally expensive; obliterating files requires that the entire body of metadata be scanned per file argument. Avoid using `p4 obliterate` during peak usage periods.

Warning! Do not use operating system commands (`erase`, `rm`, and their equivalents) to remove files from the Perforce server root by hand.

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`*.

To destroy only one revision of a file, 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, use:

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

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

The safest way to use `p4 obliterate` is to use it ***without*** the `-y` flag until you are certain the files and revisions are correctly specified.

Deleting changelists and editing changelist descriptions

Perforce administrators can use the `-f` (force) flag with `p4 change` to change the description, date, or user name of a submitted changelist. The syntax is `p4 change -f changenumber`. This command presents the standard changelist form, but also enables superusers to edit the changelist's time, description, date, and associated user name.

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.*

Verifying files by signature

Perforce administrators can use the `p4 verify filenames` command to validate stored MD5 digests of each revision of the named files. The signatures created when users store files in the depot 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. If a new signature does not match the signature in the Perforce database for that file revision, Perforce displays the characters `BAD!` after the signature.

It is good practice to run `p4 verify` before performing your nightly system backups, and to proceed with the backup only if `p4 verify` reports no corruption.

For large installations, `p4 verify` can take some time to run. Furthermore, the database is locked when `p4 verify` is running, which prevents most other Perforce commands from being used. Administrators of large sites might want to perform `p4 verify` on a weekly basis, rather than a nightly basis.

If you ever see a `BAD!` signature during a `p4 verify` command, your database or versioned files might be corrupt, and you should contact Perforce Technical Support.

Verifying files during server upgrades

It is good practice to use `p4 verify` as follows before and after server upgrades:

1. Before the upgrade, run:

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

   to verify the integrity of your server before the upgrade.

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 new system.

Defining filetypes with p4 typemap

By default, Perforce automatically determines if a file is of type `text` or `binary` based on an analysis of the first 8192 bytes of a file. If the high bit is clear in each of the first 8192 bytes, Perforce assumes it to be `text`; otherwise, it is assumed to be `binary`.

Although this default behavior can be overridden by the use of the `-t` *`filetype`* flag, it's easy for users to overlook this consideration, particularly in cases where files' types are usually (but not always) detected correctly. Certain file formats, such as RTF (Rich Text Format) and Adobe PDF (Portable Document Format), can start with a series of comment fields or other textual data. If these comments are sufficiently long, such files can be erroneously detected by Perforce as being of type `text`.

The `p4 typemap` command solves this problem by enabling system administrators to set up a table that links Perforce file types with filename specifications. If an entry in the typemap table matches a file being added, it overrides the file type that would otherwise be assigned by the Perforce client program. For example, to treat all PDF and RTF files as `binary`, use `p4 typemap` to modify 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 in the mapping. The fourth period and the file extension specify that the specification applies to files ending in `.pdf` (or `.rtf`).

The following table lists recommended Perforce file types and modifiers for common file extensions.

File type	Perforce file type	Description
.asp	text	Active server page file
.avi	binary+F	Video for Windows file
.bmp	binary	Windows bitmap file
.btr	binary	Btrieve database file
.cnf	text	Conference link file
.css	text	Cascading style sheet file
.doc	binary	Microsoft Word document
.dot	binary	Microsoft Word template
.exp	binary+w	Export file (Microsoft Visual C++)
.gif	binary+F	GIF graphic file
.htm	text	HTML file
.html	text	HTML file
.ico	binary	Icon file
.inc	text	Active Server include file
.ini	text+w	Initial application settings file
.jpg	binary	JPEG graphic file
.js	text	JavaScript language source code file
.lib	binary+w	Library file (several programming languages)
.log	text+w	Log file
.mpg	binary+F	MPEG video file
.pdf	binary	Adobe PDF file
.pdm	text+w	Sybase Power Designer file
.ppt	binary	Microsoft PowerPoint file
.xls	binary	Microsoft Excel file
.zip	binary+F	ZIP archive file

Use the following `p4 typemap` table to map all of the file extensions to the Perforce file types recommended in the preceding table.

```
# Perforce File Type Mapping Specifications.
#
#  TypeMap:     a list of filetype mappings; one per line.
#               Each line has two elements:
#               Filetype: The filetype to use on 'p4 add'.
#               Path:     File pattern which will use this filetype.
# See 'p4 help typemap' for more information.

TypeMap:

        text //....asp
        binary+F //....avi
        binary //....bmp
        binary //....btr
        text //....cnf
        text //....css
        binary //....doc
        binary //....dot
        binary+w //....exp
        binary+F //....gif
        text //....htm
        text //....html
        binary //....ico
        text //....inc
        text+w //....ini
        binary //....jpg
        text //....js
        binary+w //....lib
        text+w //....log
        binary+F //....mpg
        binary //....pdf
        text+w //....pdm
        binary //....ppt
        binary //....xls
        binary+F //....zip
```

If a file type requires the use of more than one file type modifier, specify the modifiers consecutively. For example, `binary+lFS10` refers to a `binary` file with exclusive-open (`l`), stored in full (`F`) rather than compressed, and for which only the most recent ten revisions are stored (`S10`).

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

Implementing sitewide pessimistic locking with p4 typemap

By default, Perforce supports concurrent development, but environments in which only one person is expected to have a file open for edit at a time can implement pessimistic locking by using the `+l` (exclusive open) modifier as a partial filetype. If you use the following typemap, the `+l` modifier is automatically applied to all newly added files in the depot:

```
Typemap:
        +l //depot/...
```

If you use this typemap, any files your users add to the depot after you update your typemap automatically have the `+l` modifier applied, and may only be opened for edit by one user at a time. The typemap table applies only to new additions to the depot; after you update the typemap table for sitewide exclusive open, files previously submitted without `+l` must be opened for edit with `p4 edit -t+l` *`filename`* and resubmitted. Similarly, users with files already open for edit must update their filetypes with `p4 reopen -t+l` *`filename`*.

Forcing operations with the -f flag

Certain commands support the `-f` flag, which enables Perforce administrators and superusers to force certain operations unavailable to ordinary users. Perforce administrators can use this flag with `p4 branch`, `p4 change`, `p4 client`, `p4 job`, `p4 label`, and `p4 unlock`. Perforce superusers can also use it to override the `p4 user` command. 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 mapping
	`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

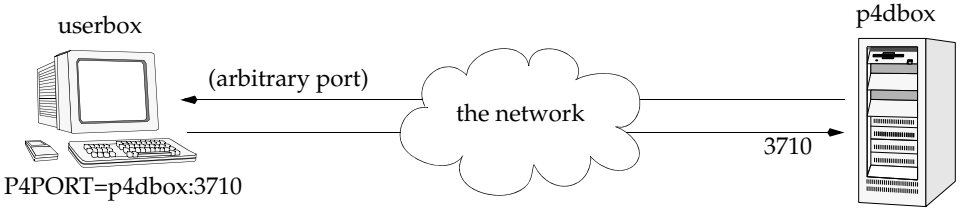
Command	Syntax	Function
`p4 client`	`p4 client -f clientname`	Allows the modification date to be changed while editing the client specification
	`p4 client -f -d clientname`	Deletes the client, ignoring ownership, even if the client has opened files
`p4 job`	`p4 job -f [jobname]`	Allows the manual update 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 This command requires `super` access.
	`p4 user -f -d username`	Deletes the user, ignoring ownership This command requires `super` access.

Advanced Perforce administration

Running Perforce through a firewall

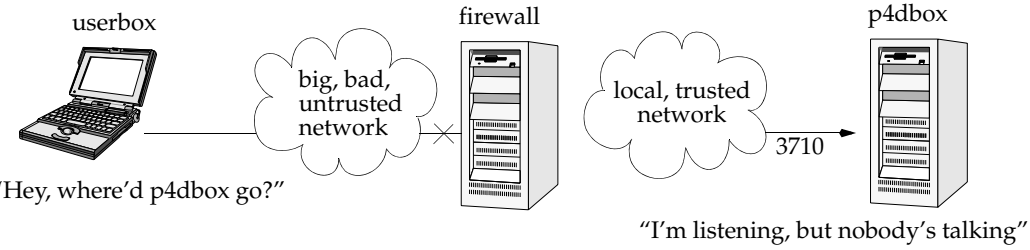
Perforce clients communicate with a Perforce server using TCP/IP. The server listens for connections at a specified port on the machine on which it's running, and clients make connections to that port.

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 that 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`).

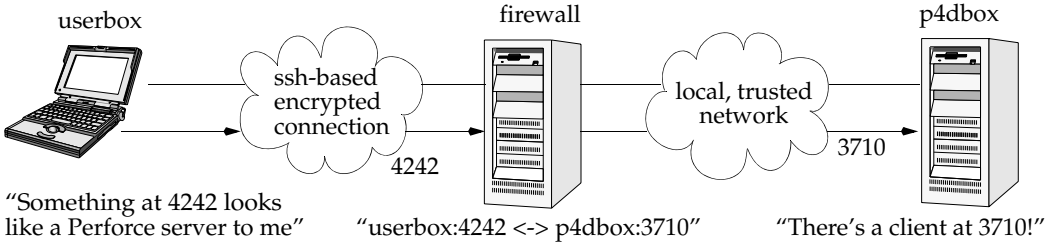
The OpenSSH FAQ can also be found online at the main site (`http://www.openssh.com/faq.html`).

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 might 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 the following command:

```
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* permit 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 you specify both an IP address *and* a port number in `P4PORT` when starting `p4d`, the Perforce server 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 configure `p4d` to listen only to a specific network interface or IP address, you can force your Perforce server to ignore all nonlocal 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 that waits for connections from clients. To have `p4d` start up only when connections are made to it, using `inetd` and `p4d -i`, add the following line to `/etc/inetd.conf`:

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

and then add the following line 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

The "extra" `p4d` on the `/etc/inetd.conf` line must be present; `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 that invokes several thousand `p4` commands, each of which spawns an instance of `p4d` via `inetd` can cause `inetd` to temporarily disable the service. Depending on your system, you might 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 multiplatform 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/file.c` and `//depot/MAIN/FILE.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.

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 that users can encounter when sharing development projects across UNIX and Windows.

If you are running a pre-97.2 server on Windows, please contact `support@perforce.com` to discuss upgrading your server and database.

For current releases of the server:

- 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.

The following table summarizes these rules.

Case-sensitive	UNIX server	Windows server
Pathnames and filenames	Yes	No
Database entities (workspaces, labels, and so on.)	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 filenames and pathnames when adding new files. They might not realize that files added as `//depot/main/one.c` and `//depot/MAIN/two.c` will appear in two different directories when synced to 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 example, 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 dir/file1
p4 add dir/file2
p4 add DIR/file3
```

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

Monitoring server activity

Use the `p4 monitor` command to obtain information about Perforce-related processes running on your Perforce server machine.

Enabling process monitoring

Server process monitoring requires minimal system resources, but you must enable process monitoring for `p4 monitor` to work. To enable process monitoring, set the `monitor` counter as follows:

```
p4 counter -f monitor 1
```

After you set or change the `monitor` counter, you must stop and restart the Perforce server for your change to take effect.

Enabling idle processes monitoring

By default, `IDLE` processes (often associated with custom applications based on the Perforce API) are not included in the output of `p4 monitor`. To include idle processes in the output of `p4 monitor`, use monitoring level 2.

```
p4 counter -f monitor 2
```

You must stop and restart the Perforce server for your change to take effect.

Listing running processes

To list the processes running on the Perforce server, use the command:

```
p4 monitor show
```

By default, each line of `p4 monitor` output looks like this:

```
pid status owner hh:mm:ss command [args]
```

where *pid* is the UNIX process ID (or Windows thread ID), *status* is `R` or `T` depending on whether the process is running or marked for termination, *owner* is the Perforce user name of the user who invoked the command, *hh:mm:ss* is the time elapsed since the command was called, and *command* and *args* are the command and arguments as received by the Perforce server. For example:

```
$ p4 monitor show
74612 R qatool     00:00:47 job
78143 R edk        00:00:01 filelog
78207 R p4admin    00:00:00 monitor
```

To show the arguments with which the command was called, use the `-a` (arguments) flag:

```
$ p4 monitor show -a
74612 R qatool     00:00:48 job job004836
78143 R edk        00:00:02 filelog //depot/main/src/proj/file1.c //dep
78208 R p4admin    00:00:00 monitor show -a
```

To obtain more information about user environment, use the `-e` flag. The `-e` flag produces output of the form:

```
pid client IP-address status owner workspace hh:mm:ss command [args]
```

where *client* is the Perforce client program (and version string or API protocol level), *IP-address* is the IP address of the user's Perforce client program, and *workspace* is the name of the calling user's current client workspace setting. For example:

```
$ p4 monitor show -e
74612 p4/2005.2 192.168.10.2   R qatool    buildenvir 00:00:47 job
78143           192.168.10.4   R edk       eds_elm    00:00:01 filelog
78207 p4/2005.2 192.168.10.10  R p4admin   p4server   00:00:00 monitor
```

By default, all user names and (if applicable) client workspace names are truncated at 10 characters, and lines are truncated at 80 characters. To disable truncation, use the `-l` (long-form) option:

```
$ p4 monitor show -a -l
74612 R qatool     00:00:50 job job004836
78143 R edk        00:00:04 filelog //depot/main/src/proj/file1.c //dep
ot/main/src/proj/file1.mpg
78209 R p4admin    00:00:00 monitor show -a -l
```

Only Perforce administrators and superusers can use the `-a`, `-l`, and `-e` options.

Marking processes for termination

If a process on a Perforce Server is consuming excessive resources, administrators and superusers can mark it for termination with `p4 monitor terminate`.

Once marked for termination, the process is terminated by the Perforce server within 50000 scan rows or lines of output. Only processes that have been running for at least ten seconds can be marked for termination.

Users of terminated processes are notified with the following message:

```
Command has been canceled, terminating request
```

Processes that involve the use of interactive forms (such as `p4 job` or `p4 user`) can also be marked for termination, but data entered by the user into the form is preserved. Some commands, such as `p4 obliterate`, cannot be terminated.

Clearing entries in the process table

Under some circumstances (for example, a Windows machine is rebooted while certain Perforce commands are running), entries may remain in the process table even after the process has terminated.

Perforce administrators and superusers can remove these erroneous entries from the process table altogether with `p4 monitor clear` *`pid`*, where *`pid`* is the erroneous process ID. To clear all processes from the table (running or not), use `p4 monitor clear all`.

Running processes removed from the process table with `p4 monitor clear` continue to run to completion.

Perforce server trace and tracking flags

To turn on command tracing or performance tracking, specify the appropriate `-v` *`subsystem=value`* flag to the `p4d` startup command. Use `P4LOG` or the `-L` *`logfile`* flag to specify the log file. For instance:

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

Before you activate logging, make sure that you have adequate disk space.

Windows When running Perforce as a Windows service, use the `p4 set` command to set `P4DEBUG` as a registry variable. You can also set these trace flags when running `p4d.exe` as a server process from the command line.

Setting server debug levels on a Perforce server (`p4d`) has no effect on the debug level of a Perforce Proxy (`p4p`) process, and vice versa.

In most cases, the Perforce server command tracing and tracking flags are useful only to system administrators working with Perforce Technical Support to diagnose or investigate a problem.

Command tracing

The server command trace flags and their meanings are as follows.

Trace flag	Meaning
`server=1`	Logs server commands to the server log file. (Requires server at release 98.1 or higher)
`server=2`	In addition to data logged at level 1, logs server command completion and basic information on CPU time used. Time elapsed is reported in seconds. On UNIX, CPU usage (system and user time) is reported in milliseconds, as per `getrusage()`. (Requires server at release 2001.1 or higher)
`server=3`	In addition to data logged at level 2, adds usage information for compute phases of `p4 sync` and `p4 flush` (`p4 sync -k`) commands. (Requires server at release 2001.2 or higher)

For command tracing, output appears in the specified log file, showing the date, time, username, IP address, and command for each request processed by the server.

Performance tracking

The Perforce Server produces diagnostic output in the server log whenever user commands exceed certain predetermined thresholds of resource usage. Performance tracking is enabled by default, and if `P4DEBUG` is unset (or the tracking flag is not specified on the command line), the tracking level is computed based on the number of users in the license file.

Tracking flag	Meaning
`track=0`	Turn off tracking.
`track=1`	Track all commands.
`track=2`	Track excess usage for a server with less than 10 users.
`track=3`	Track excess usage for a server with less than 100 users.
`track=4`	Track excess usage for a server with less than 1000 users.
`track=5`	Track excess usage for a server with more than 1000 users.

The precise format of the tracking output is undocumented and subject to change.

Auditing user file access

As of Release 2006.1, the Perforce Server can log individual file accesses to an audit logfile.

Auditing is disabled by default, and is only enabled if `P4AUDIT` is set to point to the location of the audit log file, or if the server is started with the `-A` *`auditlog`* flag.

When auditing is enabled, the server adds a line to the audit log file every time file content is transferred from the server to the client. On an active server, the audit log file will grow very quickly.

Lines in the audit log appear in the form:

`date time user@client clientIP command file#rev`

For example:

```
$ tail -2 auditlog
2006/05/09 09:52:45 karl@nail 192.168.0.12 diff //depot/src/x.c#1
2006/05/09 09:54:13 jim@stone 127.0.0.1 sync //depot/inc/file.h#1
```

If a command is run on the machine that runs the Perforce Server, the *`clientIP`* is shown as `127.0.0.1`.

Moving a Perforce server to a new machine

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

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

Additional considerations apply if the new machine has a different IP address/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.

For more about moving a Perforce server from one machine to another, see also the Perforce Tech Note at:

```
http://www.perforce.com/perforce/technotes/note010.html
```

Moving between machines of the same architecture

If the architecture of the two machines is the same (for example, SPARC/SPARC, or 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, it's always good practice to verify, checkpoint, and back up your system before any migration and to perform a subsequent verification after the migration.)

Moving between different architectures that use the same text format

If the internal data representation (big-endian vs. little-endian) convention differs between the two machines (for example, 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 re-create the database on the new machine. The checkpoint is a text file that can be read by a Perforce server on any architecture. For more details, see "Creating a checkpoint" on page 26.

After you create 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 re-created 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 re-created 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. Re-create 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.

Moving between Windows and UNIX

In this case, both the architecture of the system *and* the CR/LF text file convention are different. You still have to create a checkpoint, copy it, and re-create the database on the new platform, but when you move the depot subdirectories containing your versioned files, you 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`") need to be transferred in different ways 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.

Warning Windows is a case-insensitive operating system. Files that differ by case only on a UNIX server will occupy the same namespace when transferred to a Windows machine. For instance, files `Makefile` and file `makefile` on a UNIX server will appear to be the same file on a Windows machine.

Due to the risk of data loss due to case collision, migrations from UNIX servers to Windows are not supported.

Contact Perforce Technical Support for assistance when migrating a Perforce server from Windows to UNIX.

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 will need to update any IP-address-based protections in your protections table. See "Administering Perforce: Protections" on page 81 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 server's 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 must 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

New depots are defined with the command `p4 depot` *`depotname`*. Depots can be defined as `local`, `remote`, or `spec` depots.

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

Local depots reside on the Perforce server normally accessed by the user's Perforce client program. When using local depots, a 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 program uses the Perforce server specified by the user's `P4PORT` environment variable or equivalent setting as a means to access a second, *remote*, Perforce server. The local Perforce server communicates with the

remote Perforce server in order to access a subset of its files. Remote depots are primarily used to facilitate the sharing of code (that is, "code drops") between separate organizations, and are discussed in "Remote depots and distributed development" on page 68.

Remote depots are not a generalized solution for load-balancing or network access problems. To support shared development or to deal with load-balancing or network access problems, see "Perforce Proxy" on page 151.

The spec depot is a special case. If present, the `spec` depot tracks changes to user-edited forms such as client workspace specifications, jobs, branch mappings, and so on. There can be only one `spec` depot per server.

Naming depots

Depot names share the same namespace as branches, client workspaces, and labels. For example, `//rel2` refers uniquely to one of the depot `rel2`, the workspace `rel2`, the branch `rel2`, or the label `rel2`; you can't simultaneously have both a depot and a label named `rel2`.

Defining new local depots

To define a new local depot (that is, to create 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:    local
Suffix:     .p4s
Map:        book/...
```

The `Address:` and `Suffix:` fields do not apply to local depots and are ignored.

By default, the `Map:` field on a local depot points to a depot directory matching the depot name, relative to the server root (`P4ROOT`) setting for your server. To store a depot's versioned files on another volume or drive, specify an absolute path in the `Map:` field. This path need not be under `P4ROOT`. Absolute paths in the `Map:` field on Windows must be specified with forward slashes (for instance, `d:/newdepot/`) in the `p4 depot` form.

Enabling versioned specifications with the spec depot

In order for your users to retrieve change histories of user-edited forms, you must enable versioned specifications. The spec depot can have any name, but it must be of type `spec`, and there can only be one spec depot per server. (If you already have a spec depot, attempting to create another one results in an error message.)

After you have enabled versioned specs by creating the spec depot, all user-generated forms (such as client workspace specifications, jobs, branch mappings, and so on) are automatically archived as text files in the spec depot. Filenames within the spec depot are automatically generated by the server, and are represented in Perforce syntax as follows:

`//`*`specdepotname`*`/`*`formtype`*`/`*`objectname`*`[`*`suffix`*`]`

Creating the spec depot

To create a spec depot named `//spec`, enter `p4 depot spec`, and fill in the resulting form as follows:

```
Depot:      spec
Type:       spec
Address:    local
Map:        spec/...
Suffix:     .p4s
```

The `Address:` field does not apply to spec depots and is ignored.

Using a `Suffix:` is optional, but specifying a file extension for objects in the spec depot simplifies usability for users of graphical client programs such as P4V, because users can associate the suffix used for Perforce specifications with their preferred text editor. The default suffix for these files is `.p4s`.

For example, if you create a spec depot named `spec`, and use the default suffix of `.p4s`, your users can see the history of changes to `job000123` by using the command

```
p4 filelog //spec/job/job000123.p4s
```

or by using P4V to review changes to `job000123.p4s` in whatever editor is associated with the `.p4s` file extension on their workstation.

Populating the spec depot with current forms

After you create a spec depot, you can populate it with the `p4 admin updatespecdepot` command. This command causes the Perforce Server to archive stored forms (specifically, `client`, `depot`, `branch`, `label`, `typemap`, `group`, `user`, and `job` forms) into the spec depot.

To archive all current forms, use the `-a` flag:

```
p4 admin updatespecdepot -a
```

To populate the spec depot with only one type of form (for instance, extremely large sites might elect to update only one table at a time), use the `-s` flag and specify the form *`type`* on the command line. For example:

```
p4 admin updatespecdepot -s job
```

In either case, only those forms that have not yet been archived are added to the spec depot; after the spec depot is created, you only need to use `p4 admin updatespecdepot` once.

Listing depots

To list all depots known to the current Perforce server, use the `p4 depots` command.

Deleting depots

To delete a depot, use `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` and `spec` 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 you use `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 46.

Remote depots and distributed development

Remote depots are designed to support shared *code*, not shared *development*. They enable independent organizations with separate Perforce installations to integrate changes between Perforce installations. Briefly:

- A "remote depot" is a depot on your Perforce server of type `remote`. It acts as a pointer to a depot of type "local" that resides on a second Perforce server.
- A user of a remote depot is typically a build engineer or handoff administrator responsible for integrating software between separate organizations.
- Control over what files are available to a user of a remote depot resides with the administrator of the remote server, *not* the users of the local server.
- See "Restricting access to remote depots" on page 72 for security requirements.

When to use remote depots

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.

Partitioning joint development projects into separate Perforce installations does not improve throughput, and usually only complicates administration. If your site is engaged in distributed development (that is, developers in multiple sites working on the same body of code), it is usually preferable to set up a Perforce installation with all code in depots resident on one Perforce server, and to cache frequently accessed files at each development site with Perforce Proxy.

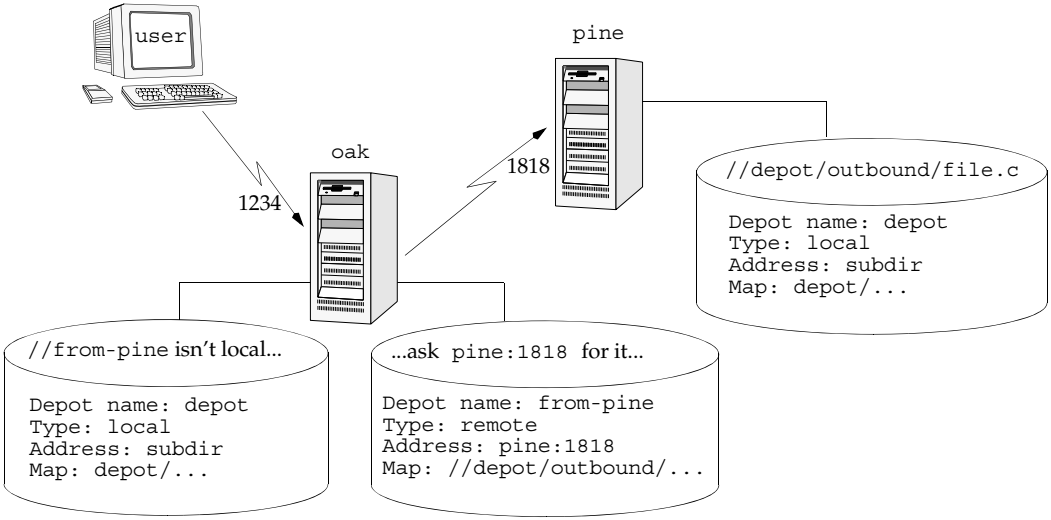
If, however, your organization regularly imports or exports material from other organizations, you might want to consider using Perforce's remote depot functionality to streamline your code drop procedures.

How remote depots work

The following diagram illustrates how Perforce client programs use a user's default Perforce server to access files in a depot hosted on another Perforce server.

In this example, an administrator of a Perforce server at `oak:1234` is retrieving a file from a remote server at `pine:1818`.

```
P4PORT=oak:1234
p4 integ //from-pine/file.c //depot/codedrops/file.c
```



Although it is possible to permit individual developers to sync files from remote depots into their client workspaces, this is generally an inefficient use of resources.

The preferred technique for using remote depots is for your organization's build or handoff administrator to integrate files from a remote depot into an area of your local depot. After the integration, your developers can access copies of the files from the local depot into which the files were integrated.

To accept a code drop from a remote depot, 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; you cannot make changes in the local branch and integrate them back into the remote depot. The copies of the files integrated into your Perforce installation become the responsibility of your site's development team; the files on the depot remain under the control of the development team at the other Perforce installation.

Restrictions on remote depots

Prior to Release 99.2, remote depots were accessible only by Perforce servers running at the same release levels. At Release 99.2 and higher, remote depots are interoperable between release levels.

Remote depots facilitate the sharing of code between organizations (as opposed to the sharing of development within a single organization). Consequently, access to remote depots is restricted to read-only operations, and server metadata (information about client workspaces, changelists, labels, and so on) cannot be accessed using remote depots.

Using remote depots for code drops

Performing a code drop requires coordination between two organizations, namely the site receiving the code drop and the site providing the code drop. In most cases, the following three things must be configured:

- The Perforce administrator at the site receiving the code drop must create a remote depot on his or her Perforce server that points to the site providing the code drop.

  This is described in "Defining remote depots" on page 71.

- The Perforce administrator at the site providing the code drop should configure his or her Perforce server to allow the recipient site's remote depot to access the providing site's Perforce server.

  This is described in "Restricting access to remote depots" on page 72.

- The configuration manager or integration manager at the receiving site must integrate the desired files from the remote depot into a local depot under his or her control.

  This is described in "Receiving a code drop" on page 73.

Defining remote depots

To define a new remote depot:

1. Create the depot with `p4 depot depotname`.
2. Set the `Type:` to `remote`.
3. Direct your Perforce server to contact the remote Perforce server by providing the remote server's name and listening port in the `Address:` field.

   A remote server's host and port are specified in the `Address:` field just as though it were a `P4PORT` setting.
4. Set the `Map:` field to map into the desired portion of the remote server's namespace.

   For remote depots, the mapping contains a subdirectory relative to the remote depot namespace. For example, `//depot/outbound/...` maps to the `outbound` subdirectory of the depot named `depot` hosted on the remote server.

   The `Map:` field must contain a single line pointing to this subdirectory, specified in depot syntax, and containing the "`...`" wildcard on its right side.

   If you are unfamiliar with client views and mappings, see the *P4 User's Guide* for general information about how Perforce mappings work.
5. The `Suffix:` field does not apply to remote depots; ignore this field.

In order for anyone on your site to access files in the remote depot, the administrator of the remote server must grant `read` access to user `remote` to the depots and subdirectories within the depots specified in the `Map:` field.

Example: *Defining a remote depot*

Lisa is coordinating a project and wants to provide a set of libraries to her developers from a third-party development shop. The third-party development shop uses a Perforce server on host `pine` *that listens on port* `1818`*. Their policy is to place releases of their libraries on their server's single depot* `depot` *under the subdirectory* `outbound`*.*

Lisa creates a new depot from which she can access the code drop; she'll call this depot `from-pine`*; she'd type* `p4 depot from-pine` *and fill in the form as follows:*

```
Depot:      from-pine
Type:       remote
Address:    pine:1818
Map:        //depot/outbound/...
```

This creates a remote depot called `from-pine` *on Lisa's Perforce server; this depot (*`//from-pine`*) maps to the third party's* `depot`*'s namespace under its* `outbound` *subdirectory.*

Restricting access to remote depots

Remote depots are always accessed by a virtual user named `remote`. This virtual user does not consume a Perforce license.

By default, all the files on any Perforce server can 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. Perforce recommends that administrators deny access to user `remote` across all files and all depots by adding the following permission line in the `p4 protect` table:

```
list user remote *  -//...
```

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

Example security configuration

Using the two organizations described in "Receiving a code drop" on page 73, a basic set of security considerations for each site would include:

On the local (`oak`) site:

- Deny access to `//from-pine` to all users. Developers at the `oak` site have no need to access files on the `pine` server by means of the remote depot mechanism.
- Grant `read` access to `//from-pine` to your integration or build managers. The only user at the `oak` site who requires access the `//from-pine` remote depot is the user (in this example, `adm`) who performs the integration from the remote depot to the local depot.

  To accomplish this, the `oak` Perforce administrator should include the following lines to the `p4 protect` table:

  ```
  list user * * -//from-pine/...
  read user adm * //from-pine/...
  ```

On the remote (`pine`) site, access to code residing on `pine` is entirely the responsibility of the `pine` server's administrator. At a minimum, this administrator should:

- Preemptively deny access to user `remote` across all depots from all IP addresses:

  ```
  list user remote * -//...
  ```

  Adding these lines to the `p4 protect` table is sound practice for any Perforce installation whether its administrator intends to use remote depots or not.

- Grant `read` access to user `remote` to only those areas of the `pine` server into which code drops are to be placed. In this example, outgoing code drops are published in the `//depot/outbound/...` subdirectory on the `pine` server.

- Grant `read` access for user `remote` only to the IP address of the Perforce servers authorized to receive code drops. If `oak`'s IP address is `192.168.41.2`, the `pine` Perforce administrator should add the following to the `p4 protect` table:

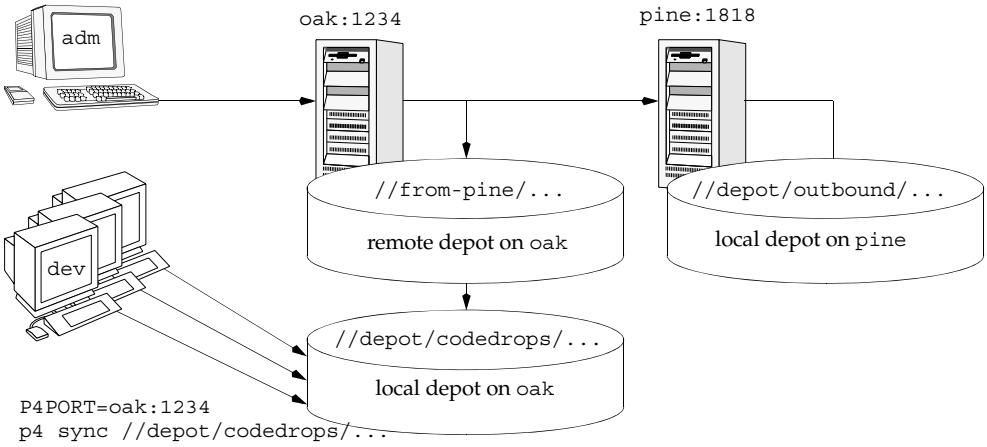
```
read user remote 192.168.41.2 //depot/outbound/...
```

Receiving a code drop

To perform a handoff or code drop between two Perforce installations:

1. Developers on `pine:1818` complete work on a body of code for delivery.
2. The build or release manager on `pine:1818` branches the deliverable code into an area of `pine:1818` intended for outbound code drops. In this example, the released code is branched to `//depot/outbound/...`
3. A Perforce administrator at `oak:1234` configures a remote depot called `//from-pine` on the `oak` server. This remote depot contains a `Map:` field that directs the `oak` server to the `//depot/outbound` area of `pine:1818`.
4. Upon notification of the release's availability, a build or release manager at `oak:1234` performs the code drop by integrating files in the `//from-pine/...` remote depot into a suitable area of the local depot, such as `//depot/codedrops/pine`.
5. Developers at `oak:1234` can now use the `pine` organization's code, now hosted locally under `//depot/codedrops/pine`. Should patches be required to `pine`'s code, `oak` developers can make such patches under `//depot/codedrops/pine`. The `pine` group retains control over its code.

```
P4PORT=oak:1234
p4 integrate //from-pine/... //depot/codedrops/pine/...
```



Managing Unicode Installations

Overview

The Perforce server can be run in Unicode mode to activate support for:

- File names or directory names that contain Unicode characters, and
- Perforce identifiers (for example, user names) and specifications (for example, changelist descriptions or jobs) that contain Unicode characters

> **Note** If you need to manage textual files that contain Unicode characters, but do not need the features listed above, you do not need to run your server in Unicode mode. For such installations, assign the Perforce `utf16` file type to textual files that contain Unicode characters.

In Unicode mode, the Perforce server also translates `unicode` files and metadata to the character set configured for the client machine, and verifies that the unicode files and metadata contain valid UTF-8 characters.

To configure a Unicode-mode server and the client machines that access it, you must:

- Put the server into Unicode mode, and
- Configure client machines to receive files and metadata correctly

The following sections tell you how to configure your Perforce installation to manage such files.

> **Note** The Perforce server limits the lengths of strings used to index job descriptions, specify filenames and view mappings, and identify client workspaces, labels, and other objects. The most common limit is 2048 bytes. Because no basic Unicode character expands to more than three bytes, you can ensure that no name exceeds the Perforce limit by limiting the length of object names and view specifications to 682 characters for Unicode-mode servers.

Configuring the Perforce Server

To enable the Perforce server to manage files and directories with names that include Unicode characters, you must configure your server to run in Unicode mode. In Unicode mode, the server:

- Supports Unicode in file and path names and Perforce metadata
- Translates metadata and `unicode` files according to the client machine's `P4CHARSET` environment variable, and
- Validates `unicode` files and metadata to ensure they are valid UTF-8

> **Note** Converting a server to Unicode mode is a one-way operation! You cannot restore a Unicode server to its previous state.

Enabling Unicode mode

To convert a server to Unicode mode, perform the following steps:

1. Stop the server by issuing the `p4 admin stop` command.
2. Create a server checkpoint, as described in the *Perforce System Administrator's Guide*.
3. Convert the server to Unicode mode by invoking the server (`p4d`) and specifying the `-xi` flag, for example:

   ```
   p4d -xi -r <p4root>
   ```

   The server verifies that its existing metadata contains only valid UTF-8 characters, then creates and sets a protected counter called "unicode," which is used as a flag to ensure that the next time you start the server, it runs in Unicode mode. After validating metadata and setting the counter, `p4d` exits and displays the following message:

   ```
   Server switched to Unicode mode.
   ```

   > **Note** If the server detects invalid characters in its metadata, it displays error messages such as the following:
   >
   > ```
   > Table db.job has 7 rows with invalid UTF8.
   > ```
   >
   > In case of such errors, contact Perforce Technical Support for instructions on locating and correcting the invalid characters.

4. Restart `p4d`, specifying server root and port as you normally do. The server now runs in Unicode mode.

Localizing server error messages

By default, the Perforce server informational and error messages are in English. You can localize server messages. To ensure best results, contact Perforce Technical Support. The following overview explains the localization process.

To localize the Perforce server messages:

1. Obtain the message file from Perforce Technical Support.
2. Edit the message file, translating messages to the target language. Each message includes a two-character language code. Change the language code from `en` to your own code for the target language. Do not change any of the key parameters or named parameters (which are specified between percent signs, for example, `%depot%`). You can change the order in which the parameters appear in the message.

   Example of correct translation:

   Original English

   ```
   @pv@ 0 @db.message@ @en@ 822220833 @Depot '%depot%' unknown - use
   'depot' to create it.@
   ```

   Correct translation to Portuguese (note reordered parameters)

   ```
   @pv@ 0 @db.message@ @pt@ 822220833 @Depot '%depot' inexistente -
   use o comando 'depot' para criar-lo.@
   ```

3. Load the translated messages into the server by issuing the following command:

   ```
   p4d -jr /fullpath/message.txt
   ```

To view localized messages, set the `P4LANGUAGE` environment variable on client machines to the language code you assigned to the messages in the translated message file. For example, to have your messages returned in Portuguese, set `P4LANGUAGE` to `pt`.

> **Note** To view localized messages using P4V, you must set the `LANG` environment variable to the language code that you use in the messages file.

Configuring Client Machines

To correctly interoperate with Unicode-mode servers, you must configure client machines by setting the `P4CHARSET` environment variable. Recommendation: use separate workspaces for connections that require different character sets.

> **Warning!** If you access your server from an incorrectly configured client machine, you risk data corruption! For example, if you submit a file using a `P4CHARSET` that is different than the one you used to sync it, the file is translated in a way that is likely to be incorrect.

Setting P4CHARSET on Windows

To set P4CHARSET for all users on a workstation, you need Perforce administrator privileges. Issue the following command:

```
p4 set -s P4CHARSET=character_set
```

To set P4CHARSET for the user currently logged in:

```
p4 set P4CHARSET=character_set
```

Your client machine must have a compatible True Type or Open Type font installed.

Setting P4CHARSET on Unix

You can set P4CHARSET from a command shell or in a startup script such as .kshrc, .cshrc, or .profile. To determine the proper value for P4CHARSET, examine the setting of the LANG or LOCALE environment variable. Common settings are as follows:

If $LANG is...	Set P4CHARSET to
en_US.ISO_8859-1	iso8859-1
ja_JP.EUC	eucjp
ja_JP.PCK	shiftjis

In general, for a Japanese installation, set P4CHARSET to eucjp, and for a European installation, set P4CHARSET to iso8859-1.

The following table lists common settings for P4CHARSET.

Client language	Client platform	P4CHARSET
Japanese	Windows (code page 932)	shiftjis
	UNIX (LOCALE setting varies)	shiftjis or eucjp
High-ASCII (recommended by ISO for use in modern European languages)	Windows (code page 1252)	winansi
	UNIX (LOCALE setting varies)	iso8859-1 or iso8859-15
	Classic MacOS (LOCALE setting varies)	macosroman
	Macintosh OS X	utf8

For details about the UTF-8 and UTF-16 variants, see "UTF character sets and Byte Order Markers (BOMs)" on page 78.

Windows: Determining your Code Page

A *code page* maps characters to values. On Windows machines, the value that you assign to `P4CHARSET` depends on the code page that your client machine uses.

To determine the code page in use on a Windows machine, either:

- **From the control panel**, launch the **Regional and Language Options** control panel. On the Languages tab, click the **Details...** button.
- **From the command prompt**, issue the `chcp.exe` command Your current code page is displayed; for example:

```
Active code page: 1252
```

Note that the DOS environment can be assigned a code page that is different from the one that is inherited by programs that are launched from the **Start** menu.

UTF character sets and Byte Order Markers (BOMs)

Byte order markers (BOMs) are used in UTF files to specify the order in which multibyte characters are stored and to identify the file content as Unicode. Not all extended-character file formats use BOMs.

To ensure that such files are translated correctly by the Perforce server when the files are synced or submitted, you must set `P4CHARSET` to the character set that corresponds to the format used on your client machine by the applications that access them, such as text editors or IDEs. Typically the formats are listed when you save the file using the **Save As...** menu option.

The following table lists valid settings for `P4CHARSET` for specifying byte order properties of UTF files.

Client Unicode format	BOM?	Big or Little-Endian	Set P4CHARSET to	Remarks
UTF-8	No	(N/A)	`utf8`	Suppresses Perforce server UTF-8 validation
	Yes		`utf8-bom`	
	No		`utf8unchecked`	
	Yes		`utf8unchecked-bom`	
UTF-16	Yes	Per client	`utf16`	Synced with a BOM according to the client platform byte order
	Yes	Little	`utf16le`	Best choice for Windows Unicode files

Client Unicode format	BOM?	Big or Little-Endian	Set P4CHARSET to	Remarks
	Yes	Big	`utf16be`	
	No	Per client	`utf16-nobom`	
	No	Little	`utf16le-nobom`	
	No	Big	`utf16be-nobom`	
UTF-32	Yes	Per client	`utf32`	Synced with a BOM according to the client platform byte order
	Yes	Little	`utf32le`	
	Yes	Big	`utf32be`	
	No	Per client	`utf32-nobom`	
	No	Little	`utf32le-nobom`	
	No	Big	`utf32be-nobom`	

If you set `P4CHARSET` to a UTF-8 setting, the Perforce server does not translate text files when you sync or submit them. Starting with release 2006.1, Perforce verifies that such files contain valid UTF-8 data.

Controlling translation of server output

If you set `P4CHARSET` to any `utf16` or `utf32` setting, you must set the `P4COMMANDCHARSET` to a non `utf16` or `utf32` character set in which you want server output displayed. "Server output" includes informational and error messages, diff output, and information returned by reporting commands.

To specify `P4COMMANDCHARSET` on a per-command basis, use the `-Q` flag. For example, to display all filenames in the depot, as translated using the `winansi` code page, issue the following command:

```
p4 -Q winansi files //...
```

Diffing files

The `p4 diff2` command, which compares two depot file revisions, can only compare files that have the same Perforce file type.

Using other Perforce client applications

If you are using other Perforce client applications, note how they handle Unicode environments:

- **P4V (Perforce Visual Client):** the first time you connect to a Unicode-mode server, you are prompted to choose the character encoding. Thereafter, P4V retains your selection in association with the connection.
- **P4Web:** when you invoke P4Web, you can specify the character encoding on the command line using the `-C` flag. P4Web uses this flag when it sends commands to a Unicode-mode server. This approach means that each instance of P4Web can handle a single character encoding and that browser machines must have compatible fonts installed.
- **P4Merge:** To configure the character encoding used by P4Merge, choose P4Merge's **File>Character Encoding...** menu option.
- **IDE plug-ins (SCC-based):** the first time you connect to a Unicode-mode server, you are prompted to choose the character encoding. Thereafter, the plug-in retains your selection in association with the connection.

Troubleshooting Client Machines in Unicode installations

To prevent file corruption, it is essential that you configure your client machine correctly. The following section describes common problems and provides solutions.

- "Cannot Translate" error message

  This message is displayed if your client machine is configured with a character set that does not include characters that are being sent to it by the Perforce server. Your client machine cannot display unmapped characters. For example, if `P4CHARSET` is set to `shiftjis` and your depot contains files named using characters from the Japanese EUC character set that do not have mappings in shift-JIS, you see the "Cannot translate" error message when you list the files by issuing the `p4 files` command.

  To ensure correct translation, do not use unmappable characters in Perforce user specifications, client specifications, jobs, or file names.

- Strange display of file content

If you attempt to display an extended-character text file and see odd-looking text, your client machine might lack the font required to display the characters in the file. Typical symptoms of this problem include the display of question marks or boxes in place of characters. To solve this problem, install the required font.

Chapter 4 Administering Perforce: Protections

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

When should protections be set?

Run `p4 protect` immediately after installing Perforce for the first time. Before the first call to `p4 protect`, every Perforce user is a superuser and thus can access and change anything in the depot. The first time a user runs `p4 protect`, a protections table is created that gives superuser access to the user from all IP addresses, and lowers all other users' access level to `write` permission on all files from all IP addresses.

The Perforce protections table is stored in the `db.protect` file in the server root directory; if `p4 protect` is first run by an unauthorized user, 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     *         195.3.24.0/24  -//...
    write    user     joe       *              -//...
    write    user     lisag     *              -//depot/...
    write    user     lisag     *              //depot/doc/...
    super    user     edk       *              //...
```

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

Note If your site is also using the Perforce Proxy, see "P4P and protections" on page 155 to learn how to configure the protections table to require users at remote sites to use the proxy.

The permission lines' five fields

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

The meanings of these fields are shown in the following table.

Field	Meaning
`Access Level`	Which access level (`list`, `read`, `open`, `write`, `review`, `admin`, or `super`) or specific right (`=read`, `=open`, `=write`, or `=branch`) is being granted or denied. Each permission level includes all the permissions above it (except for `review`). Each permission right (denoted by an `=`) only includes the specific right and not all of the lesser rights. In general, one typically grants an access level to a user or group, after which, if finer-grained control is required, one or more specific rights may then be denied.
`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 can 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`). CIDR notation is supported. The host field can also 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 be equivalent to `192.168.41.0/24`, and `*3*` would refer to any IP address with a `3` in it. Because the client's IP address is provided by the Internet Protocol itself, the host field provides as much security as is provided by the network. You cannot combine the `*` wildcard with CIDR notation, except at the start of a line when controlling proxy matching. For more about controlling access to a Perforce server via the Perforce Proxy, see "P4P and protections" on page 155.
`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 permission levels and access rights are:

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 can run those Perforce commands that are needed to read files, such as `p4 client` and `p4 sync`. The `read` permission includes `list` access.
`=read`	If this right is denied, users cannot use `p4 print`, `p4 diff`, or `p4 sync` on files.
`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 permit the user to write the files back to the depot. The `open` level is similar to `write`, except that with `open` permission, users are not permitted to run `p4 submit` or `p4 lock`. The `open` permission includes `read` and `list` access.
`=open`	If this right is denied, users cannot open files with `p4 add`, `p4 edit`, `p4 delete`, or `p4 integrate`.
`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`.
`=write`	If this right is denied, users cannot submit open files.
`=branch`	If this right is denied, users may not use files as a source for `p4 integrate`.
`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.

Level	Meaning
admin	For Perforce administrators; grants permission to run Perforce commands that affect metadata, but not server operation. Provides write and review access plus the added ability to override other users' branch mappings, client specifications, jobs, labels, and change descriptions, as well as to update the typemap table, verify and obliterate files, and customize job specifications.
super	For Perforce superusers; grants permission to run all Perforce commands. Provides write, review, and admin access plus the added ability to create depots and triggers, edit protections and user groups, delete users, reset passwords, and shut down the server.

Each Perforce command is associated with a particular minimum access level. For example, to run p4 sync or p4 print on a particular file, the user must have been granted at least read access on that file. For a full list of the minimum access levels required to run each Perforce command, see "How protections are implemented" on page 88.

The specific rights of =read, =open, =write, and =branch can be used to override the automatic inclusion of lower access levels. This makes it possible to deny individual rights without having to then re-grant lesser rights.

For example, if you want administrators to have the ability to run administrative commands, but to deny them the ability to make changes in certain parts of the depot, you could set up a permissions table as follows:

```
admin      user      joe       *              //...
=write     user      joe       *              -//depot/build/...
=open      user      joe       *              -//depot/build/...
```

In this example, user joe can perform administrative functions, which may include reading or listing files in //depot/build/..., but he is prohibited from opening files for edit (or submitting any changes he may have open.) He can, however, continue to create and modify files outside of the protected //depot/build/... area.

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 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 never need to edit those files. For example, an engineer might have write permission for source files, but have only read access to the documentation, and managers not working with code might be granted read access to all files.

Because `open` access enables local editing of files, but does not permit these files to be written to the depot, `open` access is 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 might need to change code in order to test theories as to why particular bugs occur, but these changes are not to 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 is granted until the code changes have been approved, after which time the protection level is 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 workstation 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 applies 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* `elm_proj/doc` *subdirectory but can only read other files. Lisa tries the following:*

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

She types `p4 edit //depot/elm_proj/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 //depot/elm_proj/READ.ME`, *and this command succeeds, because 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 //depot/elm_proj/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. See "How protections are implemented" on page 88 for a more detailed explanation.

Example: *Exclusionary protections*

An administrator 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.*

Which lines apply to which users or files?

Use the `p4 protects` command to display the lines from the protections table that apply to a user, group, or set of files.

With no options, `p4 protects` displays the lines in the protections table that apply to the current user. If a *`file`* argument is provided, only those lines in the protection table that apply to the named files are displayed. Using the `-m` flag displays a one-word summary of the maximum applicable access level, ignoring exclusionary mappings.

Perforce superusers can use `p4 protects -a` to see all lines for all users, or `p4 protects -u` *`user`*, `-g` *`group`*, or `-h` *`host`* flags to see lines for a specific user, group, or host IP address.

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 are assigned with `p4 protect`. Only Perforce superusers can use these commands.

Creating and editing groups

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

To add users to a group, add user names in the `Users:` field of the form generated by the `p4 group` *`groupname`* command. User names are entered under the `Users:` field header; each user name must be typed on its own line, indented. A single user can be listed in any number of groups. Group owners are not necessarily members of a group; if a group owner is to be a member of the group, the userid must also be added to the `Users:` field.

Groups can contain other groups as well as individual users. To add all users in a previously defined group to the group you're 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 are 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 can override another. For instance, if you use exclusionary mappings to deny access to an area of the depot to members of `group1`, but grant access to the same area of the depot to members of `group2`, a user who is a member of both `group1` and `group2` is either granted or denied access based on whichever line appears last in the protections table. The actual permissions granted to a specific user can be determined by replacing the names of all groups to which a particular user belongs 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; the material here is provided to explain the logic behind the behavior described above.

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

1. The command is looked up in the command access level table shown in "Access Levels Required by Perforce Commands" on page 89 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`.

2. 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 the user is permitted to know if 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, 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, the user has no permission to even list the file, and will receive a message such as `File not on client`.

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

Suppose the protections table is as follows:

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

If Ed runs `p4 print //depot/file.c`*, 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 list the file.*

3. If the first pass is successful, Perforce makes a second pass 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, and file argument, *and* has an access level greater than or equal to the access level required by the given command, 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, the user has no permission to run the command, and receives a message such as `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, because `p4 add` requires at least `open` access, you can run `p4 add` if you have `open`, `write`, `admin`, or `super` access

Command	Access Level	Notes
`add`	`open`	
`admin`	`super`	
`annotate`	`read`	
`branch`	`open`	The `-f` flag to override existing metadata or other users' data requires `admin` access.
`branches`	`list`	
`browse`	`none`	

Command	Access Level	Notes
`change`	`open`	The `-f` flag to override existing metadata or other users' data requires `admin` access.
`changes`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
`client`	`list`	The `-f` flag to override existing metadata or other users' data requires `admin` access.
`clients`	`list`	
`counter`	`review`	`list` access to at least one file in any depot is required to view an existing counter's value; `review` access is required to change a counter's value or create a new counter.
`counters`	`list`	
`dbschema`	`super`	
`delete`	`open`	
`depot`	`super`	The `-o` flag to this command, which allows the form to be read but not edited, requires only `list` access.
`depots`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
`describe`	`read`	The `-s` flag to this command, which does not display file content, requires only `list` access.
`diff`	`read`	
`diff2`	`read`	
`dirs`	`list`	
`edit`	`open`	
`export`	`super`	
`filelog`	`list`	
`files`	`list`	
`fix`	`open`	
`fixes`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
`fstat`	`list`	

Command	Access Level	Notes
group	super	The -o flag to this command, which allows the form to be read but not edited, requires only list access. The -a flag to this command requires only list access, provided that the user is also listed as a group owner.
groups	list	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
have	list	
help	none	
info	none	
integrate	open	The user must have open access on the target files and read access on the source files.
integrated	list	
job	open	The -o flag to this command, which allows the form to be read but not edited, requires only list access. The -f flag to override existing metadata or other users' data requires admin access.
jobs	list	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
jobspec	admin	The -o flag to this command, which allows the form to be read but not edited, requires only list access.
label	open	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot. The -f flag to override existing metadata or other users' data requires admin access.
labels	list	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
labelsync	list	
license	super	
lock	write	
login	list	

Command	Access Level	Notes
logout	list	
logtail	super	
monitor	list	super access is required to terminate or clear processes, or to view arguments.
move	open	
obliterate	admin	
opened	list	
passwd	list	
print	read	
protect	super	
protects	list	super access is required to use the -a, -g, and -u flags.
reopen	open	
replicate	super	
resolve	open	
resolved	open	
revert	list	
review	review	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
reviews	list	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
set	none	
shelve	write	
sizes	list	
submit	write	
sync	read	
tag	list	
tickets	none	
triggers	super	
typemap	admin	The -o flag to this command, which allows the form to be read but not edited, requires only list access.

Command	Access Level	Notes
`unlock`	`open`	The `-f` flag to override existing metadata or other users' data requires `admin` access.
`unshelve`	`write`	
`user`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
`users`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.
`verify`	`admin`	
`where`	`list`	This command doesn't operate on specific files. Permission is granted to run the command if the user has the specified access to at least one file in any depot.

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

Some commands (for example, `p4 change`, when you edit a previously submitted changelist) take a `-f` flag that can only be used by Perforce superusers. See "Forcing operations with the -f flag" on page 51 for details.

Chapter 5 Customizing Perforce: Job Specifications

Perforce's jobs feature enables users to link changelists to enhancement requests, problem reports, and other user-defined tasks. Perforce also offers P4DTG (Perforce Defect Tracking Gateway) and P4DTI (Perforce Defect Tracking Integration) as tools for integrating third-party defect tracking tools with Perforce. See "Working with third-party defect tracking systems" on page 103 for details.

The Perforce user's use of `p4 job` is discussed in the *P4 User's Guide*. This chapter covers administrator 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 might be insufficient. To modify the job template, use the `p4 jobspec` command. You must be a Perforce administrator to use `p4 jobspec`.

This chapter discusses the mechanics of altering the Perforce job template.

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, consider the default Perforce job template. The examples that follow in this chapter are based on modifications to the this 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:   2008/06/03 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 always
      105 Description text 0 required
Values:
      Status open/suspended/closed
Presets:
      Status open
      User $user
      Date $now
      Description $blank
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.
```

The job template's fields

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

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. Field names must not contain spaces.
`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.

Field / Field Type	Meaning
`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. They are also used by P4V, the Perforce Visual Client, to display tooltips.

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 includes these fields:

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

Warning Do not attempt to change, rename, or redefine fields 101 through 105. Fields 101 through 105 are used by Perforce and should not be deleted or changed. Use `p4 jobspec` only to add new fields (106 and above) to your jobs.

- Field 101 is required by Perforce and cannot be renamed nor deleted.
- Fields 102 through 105 are reserved for use by Perforce client programs. Although it is possible to rename or delete these fields, it is highly undesirable to do so. Perforce client programs might continue to set the value of field 102 (the `Status:` field) to `closed` (or some other value defined in the `Presets:` for field 102) upon changelist submission, even if the administrator has redefined field 102 for use as a field that does not contain `closed` as a permissible value, leading to unpredictable and confusing results.

Each field must be listed on a separate line. A field comprises the following 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 `106` and `199`.
`Name`	The name of the field as it should appear on the `p4 job` form. No spaces are permitted.
`Datatype`	One of five datatypes (`word`, `text`, `line`, `select`, or `date`), as described in the next table.
`Length`	The recommended size of the field's text box as displayed in P4V, the Perforce Visual 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 it 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:` the 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.

Fields have the following five datatypes.

Field Type	Explanation	Example
`word`	A single word (a string without spaces).	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.`
`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`
`bulk`	A block of text that can span multiple lines, but which is not indexed for searching with `p4 jobs -e`.	Alphanumeric data for which text searches are not expected.

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
```

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.

The following 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

Variable	Value
`$now`	The date and time at the moment the job is saved
`$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
```

Using Presets: to change default fix status

The `Presets:` entry for the job status field (field 102) has a special syntax for providing a default fix status for `p4 fix`, `p4 change -s`, and `p4 submit -s`.

To change the default fix status from `closed` to some other *`fixStatus`* (assuming that your preferred *`fixStatus`* is already defined as a valid `select` setting in the `Values:` field), use the following syntax:

```
Presets:
      Status openStatus,fix/fixStatus
```

In order to change the default behavior of `p4 fix`, `p4 change`, and `p4 submit` to leave job status unchanged after fixing a job or submitting a changelist, use the special *`fixStatus`* of `same`. For example:

```
Presets:
      Status open,fix/same
```

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.
```

These fields are also used by P4V, the Perforce Visual Client, to display tooltips.

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 you attempt to edit a job specification.

- Do not attempt to change, rename, or redefine fields 101 through 105. These fields are used by Perforce and should not be deleted or changed. Use `p4 jobspec` only to add new fields (106 and above) to your jobs.

  Field 101 is required by Perforce and cannot be renamed nor deleted.

  Fields 102 through 105 are reserved for use by Perforce client programs. Although it is possible to rename or delete these fields, it is highly undesirable to do so. Perforce client programs may continue to set the value of field 102 (the `Status:` field) to `closed` (or some other value defined in the `Presets:` for field 102) upon changelist submission, even if the administrator has redefined field 102 for use as a field that does not contain `closed` as a permissible value, leading to unpredictable and confusing results.

- After a field has been created and jobs have been entered, do not change the field's ID number. 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 field name in other `p4 jobspec` fields that reference this field name. 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 also used to display tooltips in P4V, the Perforce Visual Client.

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
        103 User word 32 required
        104 Date date 20 always
        111 Type select 10 required
        112 Priority select 10 required
        113 Subsystem select 10 required
        114 Owned_by word 32 required
        105 Description text 0 required
Values:
        Status open/closed/suspended
        Type bug/sir/problem/unknown
        Priority A/B/C/unknown
        Subsystem server/gui/doc/mac/misc/unknown
Presets:
        Status open
        User setme
        Date $now
        Type setme
        Priority unknown
        Subsystem setme
        Owned_by $user
        Description $blank
Comments:
        # Custom Job fields:
        # 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.
        # 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
        # Owned_by:      Who's fixing the bug
        # Description: Comments about the job. Required.
```

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.

Running `p4 job` against the example custom jobspec displays the following job form:

```
# Custom Job fields:
# 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.
# 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
# Owned_by:      Who's fixing the bug
# Description: Comments about the job. Required.

Job:    new
Status: open
User:   setme
Type:   setme
Priority:       unknown
Subsystem:      setme
Owned_by:       edk
Description:
        <enter description here>
```

Working with third-party defect tracking systems

Perforce currently offers two independent platforms to integrate Perforce with third-party defect tracking systems. Both platforms allow information to be shared between Perforce's job system and external defect tracking systems.

P4DTG, the Perforce Defect Tracking Gateway, is a closed-source integrated platform that includes both a graphical configuration editor and a replication engine.

P4DTI, Perforce Defect Tracking and Integration, is an open-source product available under a FreeBSD-like license.

P4DTG, The Perforce Defect Tracking Gateway

The Perforce Defect Tracking Gateway (P4DTG) includes a graphical configuration editor that you can use to control the relationship between Perforce jobs and the external system. Propagation of the data between the two systems is coordinated by a replication engine. P4DTG comes with a plug-in for HP Quality Center 9.0; the plug-in supports HP Quality Center 9.0, 9.2, and 10.0.

For more information, see the product page at:

```
http://www.perforce.com/perforce/products/p4dtg.html
```

Available from this page are an overview of P4DTG's capabilities, the download for P4DTG itself, and a link to the *Perforce Defect Tracking Gateway Guide*, which describes how to install and configure the gateway to replicate data between a Perforce server and a defect tracker.

P4DTI, Perforce Defect Tracking Integration

P4DTI is open source and available under a FreeBSD-like license, and can be extended to other products; TeamTrack and Bugzilla are the first two integrations published. 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 TeamTrack and Bugzilla implementations, an overview of 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 Perforce's integrations as your 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 that 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 the standard input of a `p4 job -i` command.

  (The `-i` flag to `p4 job` is used when you want `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: Triggers and Daemons" on page 105.

Chapter 6 Scripting Perforce: Triggers and Daemons

There are two primary methods of scripting Perforce:

- Perforce *triggers* are user-written scripts that are called by a Perforce server whenever certain operations (such as changelist submissions, changes to forms, attempts by users to log in or change passwords) are performed. If the script returns a value of `0`, the operation continues; if the script returns any other value, the operation fails. Upon failure, the script's standard output (not error output) is sent to the Perforce client program as an error message.
- *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 specified way, it runs other commands. For example, a daemon might look for newly submitted changelists and send email to users interested in tracking changes to those files. Perforce provides a number of tools that make writing daemons easier.

To make use of this chapter, you must know how to write scripts.

Triggers

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

- To validate changelist contents beyond the mechanisms afforded by the Perforce protections table. For example, you can use a pre-submit trigger to ensure that whenever `file1` is submitted in a changelist, `file2` is also submitted.
- To validate file contents as part of changelist submission. For example, you can use a mid-submit trigger to ensure that, when `file1` and `file2` are submitted, both files refer to the same set of header files.
- To start build processes after successful changelist submission.
- To validate forms, or to provide customized versions of Perforce forms. For example, you can use form triggers to generate a customized default workspace view when users run the `p4 client` command, or to ensure that users always enter a meaningful workspace description.
- To configure Perforce to work with external authentication mechanisms such as LDAP or Active Directory.
- To retrieve (or generate) content from data sources archived outside of the Perforce repository.

- To notify other users of attempts to change or delete data with commands such as `p4 user` and `p4 job`, or to trigger process control tools following updates to Perforce metadata.

Note As of Release 2007.3, trigger standard output is passed to the client program on both success and failure of the trigger script. Prior to this, standard output was only passed to the client program on failure of a trigger.

Warning! When you use trigger scripts, remember that Perforce commands that write data to the depot are dangerous and should be avoided. In particular, do not run the `p4 submit` command from within a trigger script.

Warning! If you write a trigger that fires on trigger forms, and the trigger fails in such a way that the `p4 triggers` command no longer works, the only recourse is to remove the `db.triggers` file in the server root directory.

Example: *A basic trigger*

The development group wants to ensure that whenever a changelist containing an `.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, performs 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 also 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`*.*

After you write the script, add it to the trigger table by editing the `p4 triggers` *form as follows:*

```
Triggers:
  rnotes change-submit  //depot/....exe  "/usr/bin/test.pl %change%"
```

(Like other Perforce forms, indent each line under the `Triggers:` *field with tabs.)*

Whenever a changelist containing an `.exe` *file is submitted, this trigger fires. If the trigger script fails, it returns a nonzero exit status, and the user's* `submit` *fails.*

The trigger table

After you have written a trigger script, create the trigger by issuing the `p4 triggers` command. The `p4 triggers` form looks like this:

```
Triggers:
  relnotecheck change-submit //depot/bld/... "/usr/bin/rcheck.pl %user%"
  verify_jobs  change-submit //depot/...     "/usr/bin/job.py %change%"
```

As with all Perforce commands that use forms, field names (such as `Triggers:`) must be flush left (not indented) and must end with a colon, and field values (that is, the set of lines you add, one for each trigger) must be indented with spaces or tabs on the lines beneath the field name.

When testing and debugging triggers, remember that any `p4` commands invoked from within the script will run within a different environment (`P4USER`, `P4CLIENT`, and so on) than that of the calling user.

You must specify the name of the trigger script or executable in ASCII, even when the server is running in Unicode mode and passes arguments to the trigger script in UTF8.

You must be a Perforce superuser to run `p4 triggers`.

Trigger table fields

Each line in the trigger table has four fields.

Field	Meaning
name	The user-defined name of the trigger. A run of the same trigger name on contiguous lines is treated as a single trigger, so that multiple *path*s may be specified. In this case, only the *command* of the first such trigger line is used.

Field	Meaning
type	There are thirteen trigger types, divided into four subtypes: changelist submission triggers, fix triggers, form triggers, and authentication triggers. Changelist submission triggers: • `change-submit`: Execute a changelist trigger after changelist creation, but before file transfer. Trigger may not access file contents. • `change-content`: Execute a changelist trigger after changelist creation and file transfer, but before file commit. To obtain file contents, use commands such as `p4 diff2`, `p4 files`, `p4 fstat`, and `p4 print` with the revision specifier `@=`*`change`*, where *`change`* is the changelist number of the pending changelist as passed to the script in the `%changelist%` variable. • `change-commit`: Execute a changelist trigger after changelist creation, file transfer, and changelist commit. Fix triggers: The special variable `%jobs%` is available for expansion; it expands to one argument for every job listed on the `p4 fix` command line (or in the `Jobs:` field of a `p4 change` or `p4 submit` form), and must therefore be the last argument supplied to the trigger script. • `fix-add`: Execute fix trigger prior to adding a fix. • `fix-delete`: Execute fix trigger prior to deleting a fix.

Field	Meaning
	Form triggers: • `form-save`: Execute a form trigger after the form contents are parsed, but before the contents are stored in the Perforce database. The trigger cannot modify the form specified in `%formfile%` variable. • `form-out`: Execute form trigger upon generation of form to end user. The trigger can modify the form. • `form-in`: Execute form trigger on edited form before contents are parsed and validated by the Perforce server. The trigger can modify the form. • `form-delete`: Execute form trigger after the form contents are parsed, but before the form is deleted from the Perforce database. The trigger cannot modify the form. • `form-commit`: Execute form trigger after the form has been committed for access to automatically-generated fields such as jobname, dates, etc. For job forms, this trigger is run by `p4 job` as well as `p4 fix` (after the status is updated). The `form-commit` trigger has access to the new job name created by `p4 job`; any `form-in` and `form-save` triggers are run before the job name is created. For job forms, this trigger is also run by `p4 change` (if a job is added or deleted by editing the `Jobs:` field of the changelist), and `p4 submit` (for any jobs present in the `Jobs:` field of the changelist). In these cases, the special variable `%action%` is available for expansion on the job `form-commit` trigger command line. The trigger cannot modify the form. Authentication triggers: • `auth-check`: Execute an authentication check trigger to verify a user's password against an external password manager during login, or when setting a new password If an `auth-check` trigger is present, the Perforce `security` counter (and any associated password strength requirement) is ignored, as authentication is now controlled by the trigger script. • `auth-set`: Execute an authentication set trigger to send a new password to an external password manager. You must restart the Perforce server after adding an `auth-check` trigger. Archive triggers: • `archive`: Execute the script when a user accesses any file with a filetype containing the `+X` filetype modifier. The script is run once per file requested. For `read` operations, scripts should deliver the file to the user on standard output. For `write` operations, scripts receive the file on standard input.

Field	Meaning
path	For changelist submission triggers (`change-submit`, `change-content`, or `change-commit`), a file pattern in depot syntax. When a user submits a changelist that contains any files that match this file pattern, the script linked to this trigger is run. Use exclusionary mappings to prevent triggers from running on specified files. For fix triggers (`fix-add` or `fix-delete`), use `fix` as the path value. For form triggers (`form-save`, `form-out`, `form-in`, `form-commit`, or `form-delete`), the name of the type of form, (one of `branch`, `change`, `client`, `depot`, `group`, `job`, `label`, `protect`, `spec`, `triggers`, `typemap`, or `user`). For authentication triggers (`auth-check` or `auth-set`), use `auth` as the path value.
command	The command for the Perforce server to run when a matching *path* applies for the trigger type. Specify the command in a way that allows the Perforce server account to locate and run the command. The command must be quoted, and can take the variables specified below as arguments. For `change-submit` and `change-content` triggers, changelist submission continues if the trigger script exits with 0, or fails if the script exits with a nonzero value. For `change-commit` triggers, changelist submission succeeds regardless of the trigger script's exit code, but subsequent `change-commit` triggers do not fire if the script exits with a nonzero value. For `form-in`, `form-out`, `form-save`, and `form-delete` triggers, the data in the specification becomes part of the Perforce database if the script exits with 0. Otherwise, the database is not updated. The `form-commit` trigger type never rejects a change; it exists primarily so that scripts can access a job number (from the `%formname%` variable) during the process of job creation. For `fix-add` and `fix-delete` triggers, fix addition or deletion continues if the trigger script exits with 0, or fails if the script exits with a nonzero value. For `auth-check` triggers (fired by `p4 login`), the user's typed password is supplied to the trigger command as standard input. If the trigger executes successfully, the Perforce ticket is issued. The user name is available as `%user%` to be passed on the command line. For `auth-set` triggers, (fired by `p4 passwd`, but only after also passing an `auth-check` trigger check) the user's old password and new password are passed to the trigger as standard input. The user name is available as `%user%` to be passed on the command line.

Trigger script variables

Use the following variables in the `command` field to pass data to a trigger script:

Argument	Description	Available for type
`%action%`	Either null or a string reflecting an action taken to a changelist or job. For example, "`pending change 123 added`" or "`submitted change 124 deleted`" are possible `%action%` values on `change` forms, and "`job000123 created`" or "`job000123 edited`" are possible `%action%` values for `job` forms.	`form-commit`
`%changelist%` `%change%`	The number of the changelist being submitted. The abbreviated form `%change%` is equivalent to `%changelist%`. A `change-submit` trigger is passed the pending changelist number; a `change-commit` trigger receives the committed changelist number.	`change-submit`, `change-content`, `change-commit`, `fix-add`, `fix-delete`, `form-commit`
`%changeroot%`	The root path of files submitted	`change-commit`
`%client%`	Triggering user's client workspace name.	all but `archive`
`%clienthost%`	Hostname of the client.	all but `archive`
`%clientip%`	The IP address of the client.	all but `archive`
`%jobs%`	A string of job numbers, expanded to one argument for each job number specified on a `p4 fix` command or for each job number added to (or removed from) the `Jobs:` field in a `p4 submit`, or `p4 change` form.	`fix-add` `fix-delete`
`%oldchangelist%`	If a changelist is renumbered on submit, this variable contains the old changelist number.	`change-commit`
`%serverhost%`	Hostname of the Perforce server.	all but `archive`
`%serverip%`	The IP address of the server.	all but `archive`
`%serverport%`	The IP address and port of the Perforce server, in the format *`ip_address:port`*.	all but `archive`
`%serverroot%`	The `P4ROOT` directory of the Perforce server.	all but `archive`
`%user%`	Perforce username of the triggering user.	all but `archive`

Argument	Description	Available for type
`%formfile%`	Path to temporary form specification file. To modify the form from an `in` or `out` trigger, overwrite this file. The file is read-only for triggers of type `save` and `delete`.	`form-commit`, `form-save`, `form-out`, `form-in`, `form-delete`
`%formname%`	Name of form (for instance, a branch name or a changelist number).	`form-commit`, `form-save`, `form-out`, `form-delete`
`%formtype%`	Type of form (for instance, `branch`, `change`, and so on).	`form-commit`, `form-save`, `form-out`, `form-in`, `form-delete`
`%op%`	Operation: `read`, `write`, or `delete`	`archive`
`%file%`	Path of archive file based on depot's `Map:` field. If the `Map:` field is relative to `P4ROOT`, the `%file%` is a server-side path relative to `P4ROOT`. If the `Map:` field is an absolute path, the `%file%` is an absolute server-side path.	`archive`
`%rev%`	Revision of archive file	`archive`

Triggering on changelists

To configure Perforce to run trigger scripts when users submit changelists, use *changelist submission triggers:* these are triggers of type `change-submit`, `change-content`, and `change-commit`.

For changelist submission triggers, the *`path`* column of each trigger line is a file pattern in depot syntax. If a changelist being submitted contains any files in this path, the trigger fires. To prevent changes to a file from firing a trigger, use an exclusionary mapping in the path.

Change-submit triggers

Use the `change-submit` trigger type to create triggers that fire after changelist creation, but before files are transferred to the server. Because change-submit triggers fire before files are transferred to the server, these triggers cannot access file contents. Change-submit triggers are useful for integration with reporting tools or systems that do not require access to file contents.

Example: *The following* `change-submit` *trigger is an* MS-DOS *batch file that rejects a changelist if the submitter has not assigned a job to the changelist. This trigger fires only on changelist submission attempts that affect at least one file in the* `//depot/qa` *branch.*

```
@echo off

rem REMINDERS
rem - If necessary, set Perforce environment vars or use config file
rem - Set PATH or use full paths (C:\PROGRA~1\Perforce\p4.exe)
rem - Use short pathnames for paths with spaces, or quotes
rem - For troubleshooting, log output to file, for instance:
rem - C:\PROGRA~1\Perforce\p4 info >> trigger.log

if not x%1==x goto doit
echo Usage is %0[change#]

:doit
p4 describe -s %1|findstr "Jobs fixed...\n\n\t" > nul
if errorlevel 1 echo No jobs found for changelist %1
p4 describe -s %1|findstr "Jobs fixed...\n\n\t" > nul
```

To use the trigger, add the following line to your triggers table:

```
sample1   change-submit //depot/qa/...   "jobcheck.bat %changelist%"
```

Every time a changelist is submitted that affects any files under `//depot/qa`*, the* `jobcheck.bat` *file is called. If the string "*`Jobs fixed...`*" (followed by two newlines and a tab character) is detected, the script assumes that a job has been attached to the changelist and permits changelist submission to continue. Otherwise, the submit is rejected.*

The second `findstr` *command ensures that the final error level of the trigger script is the same as the error level that determines whether to output the error message.*

Change-content triggers

Use the `change-content` trigger type to create triggers that fire after changelist creation and file transfer, but prior to committing the submit to the database. Change-content triggers can access file contents by using the `p4 diff2`, `p4 files`, `p4 fstat`, and `p4 print` commands with the `@=`*`change`* revision specifier, where *`change`* is the number of the pending changelist as passed to the trigger script in the `%changelist%` variable.

Use change-content triggers to validate file contents as part of changelist submission and to abort changelist submission if the validation fails.

Example: *The following* `change-content` *trigger is a Bourne shell script that ensures that every file in every changelist contains a copyright notice for the current year.*

The script assumes the existence of a client workspace called `copychecker` *that includes all of* `//depot/src`*. This workspace does not have to be synced.*

```
#!/bin/sh
# Set target string, files to search, location of p4 executable...
TARGET="Copyright `date +%Y` Example Company"
DEPOT_PATH="//depot/src/..."
CHANGE=$1
P4CMD="/usr/local/bin/p4 -p 1666 -c copychecker"
XIT=0
echo ""
# For each file, strip off #version and other non-filename info
# Use sed to swap spaces w/"%" to obtain single arguments for "for"
for FILE in `$P4CMD files $DEPOT_PATH@=$CHANGE | \
  sed -e 's/\(.*\)\#[0-9]* - .*$/\1/' -e 's/ /%/g'`
do
  # Undo the replacement to obtain filename...
  FILE="`echo $FILE | sed -e 's/%/ /g'`"
# ...and use @= specifier to access file contents:
  # p4 print -q //depot/src/file.c@=12345
  if $P4CMD print -q "$FILE@=$CHANGE" | grep "$TARGET" > /dev/null
  then echo ""
  else
      echo "Submit fails: '$TARGET' not found in $FILE"
      XIT=1
  fi
done
exit $XIT
```

To use the trigger, add the following line to your triggers table:

```
sample2  change-content //depot/src/... "copydate.sh %change%"
```

The trigger fires when any changelist with at least one file in `//depot/src` *is submitted. The corresponding* `DEPOT_PATH` *defined in the script ensures that of all the files in the triggering changelist, only those files actually under* `//depot/src` *are checked.*

Change-commit triggers

Use the `change-commit` trigger type to create triggers that fire after changelist creation, file transfer, and changelist commission to the database. Use change-commit triggers for processes that assume (or require) the successful submission of a changelist.

Example: *A* `change-commit` *trigger that sends emails to other users who have files open in the submitted changelist.*

```
#!/bin/sh
# mailopens.sh - Notify users when open files are updated
changelist=$1
workspace=$2
user=$3
p4 fstat @$changelist,@$changelist | while read line
do
  # Parse out the name/value pair.
  name=`echo $line | sed 's/[\. ]\+\([^ ]\+\) .\+/\1/'`
  value=`echo $line | sed 's/[\. ]\+[^ ]\+ \(.\+\)/\1/'`
  if [ "$name" = "depotFile" ]
  then
    # Line is "... depotFile <depotFile>". Parse to get depotFile.
    depotfile=$value
  elif [ "`echo $name | cut -b-9`" = "otherOpen" -a \
    "$name" != "otherOpen" ]
  then
    # Line is "... ... otherOpen[0-9]+ <otherUser@otherWorkspace>".
    # Parse to get otherUser and otherWorkspace.
    otheruser=`echo $value | sed 's/\(.\+\)@.\+/\1/'`
    otherworkspace=`echo $value | sed 's/.\+@\(.\+\)/\1/'`
    # Get email address of the other user from p4 user -o.
    othermail=`p4 user -o $otheruser | grep Email: \
      | grep -v \# | cut -b8-`
    # Mail other user that a file they have open has been updated
    mail -s "$depotfile was just submitted" $othermail <<EOM
The Perforce file: $depotfile
was just submitted in changelist $changelist by Perforce user $user
from the $workspace workspace.  You have been sent this message
because you have this file open in the $otherworkspace workspace.
EOM
  fi
done
exit 0
```

To use the trigger, add the following line to your triggers table:

```
sample3  change-commit //... "mailopens.sh %change% %client% %user%"
```

Whenever a user submits a changelist, any users with open files affected by that changelist receive an email notification.

Triggering on fixes

To configure Perforce to run trigger scripts when users add or delete fixes from changelists, use *fix triggers:* these are triggers of type `fix-add` and `fix-delete`.

Fix-add and fix-delete triggers

Example: *The following script, when copied to* `fixadd.sh` *and* `fixdel.sh`*, fires when users attempt to add or remove fix records, whether by using the* `p4 fix` *command, or by modifying the* `Jobs:` *field of the forms presented by the* `p4 change` *and* `p4 submit` *commands.*

```
#!/bin/bash
# fixadd.sh, fixdel.sh - illustrate fix-add and fix-delete triggers

COMMAND=$0
CHANGE=$1
NUMJOBS=$(($# - 1 ))

echo $COMMAND: fired against $CHANGE with $NUMJOBS job arguments.
echo $COMMAND: Arguments were: $*
```

These `fix-add` *and* `fix-delete` *triggers fire whenever users attempt to add (or delete) fix records from changelists. To use the trigger, add the following lines to the trigger table:*

```
sample4   fix-add    fix "fixadd.sh %change% %jobs%"
sample5   fix-delete fix "fixdel.sh %change% %jobs%"
```

Using both copies of the script, observe that `fixadd.sh` *is triggered by* `p4 fix`*, the* `fixdel.sh` *script is triggered by* `p4 fix -d`*, and either script may be triggered by manually adding (or deleting) job numbers from within the* `Jobs:` *field in a changelist form - either by means of* `p4 change` *or as part of the* `p4 submit` *process.*

Because the `%jobs%` *variable is expanded to one argument for every job listed on the* `p4 fix` *command line (or in the* `Jobs:` *field of a* `p4 change` *or* `p4 submit` *form), it must be the last argument supplied to any* `fix-add` *or* `fix-delete` *trigger script.*

Triggering on forms

To configure Perforce to run trigger scripts when users edit forms, use *form triggers:* these are triggers of type `form-save`, `form-in`, `form-out`, `form-delete`, and `form-commit`.

Use form triggers to generate customized field values for users, to validate forms, to notify other users of attempted changes to form data, and to otherwise interact with process control and management tools.

Form-save triggers

Use the `form-save` trigger type to create triggers that fire when users send changed forms to the server. Form-save triggers are called after the form has been parsed by the server but before the changed form is stored in the Perforce metadata.

Example: *To prohibit certain users from modifying their client workspaces, add the users to a group called* `lockedws` *and use the following* `form-save` *trigger.*

This trigger denies attempts to change client workspace specifications for users in the `lockedws` *group, outputs an error message containing the user name, IP address of the user's workstation, and the name of the workspace on which a modification was attempted, and notifies an administrator.*

```
#!/bin/bash
NOAUTH=lockedws
USERNAME=$1
WSNAME=$2
IPADDR=$3

GROUPS=`p4 groups "$1"`

if echo "$GROUPS" | grep -qs $NOAUTH
then
   echo "$USERNAME ($IPADDR) in $NOAUTH may not change $WSNAME"
   mail -s "User $1 workspace mod denial" admin@127.0.0.1
   exit 1
else
   exit 0
fi
```

This `form-save` *trigger fires on* `client` *forms only. To use the trigger, add the following line to the trigger table:*

```
sample6   form-save  client  "ws_lock.sh %user% %client% %clientip%"
```

Users whose names appear in the output of `p4 groups lockedws` *have changes to their client workspaces parsed by the server, and even if those changes are syntactically correct, the attempted change to the workspace is denied, and an administrator is notified of the attempt.*

Form-out triggers

Use the `form-out` trigger type to create triggers that fire whenever the Perforce server generates a form for display to the user.

Warning! Never use a Perforce command in a `form-out` trigger that fires the same `form-out` trigger, or infinite recursion will result. For example, never run `p4 job -o` from within a `form-out` trigger script that fires on `job` forms.

Example: *The default Perforce client workspace view maps the entire depot* `//depot/...` *to the user's client workspace. To prevent novice users from attempting to sync the entire depot, this Perl script changes a default workspace view of* `//depot/...` *in the* `p4 client` *form to map only the current release codeline of* `//depot/releases/main/...`

```
#!/usr/bin/perl
# default_ws.pl - Customize the default client workspace view.
$p4 = "p4 -p localhost:1666";
$formname = $ARGV[0];  # from %formname% in trigger table
$formfile = $ARGV[1];  # from %formfile% in trigger table

# Default server-generated workspace view and modified view
# (Note: this script assumes that //depot is the only depot defined)
$defaultin = "\t//depot/... //$formname/...\n";
$defaultout = "\t//depot/releases/main/... //$formname/...\n";

# Check "p4 clients": if workspace exists, exit w/o changing view.
open CLIENTS, "$p4 clients |" or die "Couldn't get workspace list";
while ( <CLIENTS> )
{
        if ( /^Client $formname .*/ ) { exit 0; }
}

# Build a modified workspace spec based on contents of %formfile%
$modifiedform = "";
open FORM, $formfile or die "Trigger couldn't read form tempfile";
while ( <FORM> )
{       ## Do the substitution as appropriate.
        if ( m:$defaultin: ) { $_ = "$defaultout"; }
        $modifiedform .= $_;
}

# Write the modified spec back to the %formfile%,
open MODFORM, ">$formfile" or die "Couldn't write form tempfile";
print MODFORM $modifiedform;
exit 0;
```

This `form-out` *trigger fires on* `client` *workspace forms only. To use the trigger, add the following line to the trigger table:*

```
sample7   form-out   client   "default_ws.pl %formname% %formfile%"
```

New users creating client workspaces are presented with your customized default view.

Form-in triggers

Use the `form-in` trigger type to create triggers that fire when a user attempts to send a form to the server, but before the form is parsed by the Perforce server.

Example: *All users permitted to edit jobs have been placed in a designated group called* `jobbers`*. The following Python script runs* `p4 group -o jobbers` *with the* `-G` *(Python marshaled objects) flag to determine if the user who triggered the script is in the* `jobbers` *group.*

```
import sys, os, marshal

# Configure for your environment
tuser = "triggerman"   # trigger username
job_group = "jobbers"  # Perforce group of users who may edit jobs

# Get trigger input args
user = sys.argv[1]

# Get user list
# Use global -G flag to get output as marshaled Python dictionary
CMD = "p4 -G -u %s -p 1666 group -o %s" % \
       (tuser, job_group)
result = {}
result = marshal.load(os.popen(CMD, 'r'))

job_users = []
for k in result.keys():
        if k[:4] == 'User': # user key format: User0, User1, ...
                u = result[k]
                job_users.append(u)

# Compare current user to job-editing users.
if not user in job_users:
        print "\n\t>>> You don't have permission to edit jobs."
        print "\n\t>>> You must be a member of '%s'.\n" % job_group
        sys.exit(1)
else: # user is in job_group -- OK to create/edit jobs
        sys.exit(0)
```

This `form-in` *trigger fires on* `job` *forms only. To use the trigger, add the following line to the trigger table:*

```
sample8   form-in  job  "python jobgroup.py %user%"
```

If the user is in the `jobbers` *group, the* `form-in` *trigger succeeds, and the changed job is passed to the Perforce server for parsing. Otherwise, an error message is displayed, and changes to the job are rejected.*

Form-delete triggers

Use the `form-delete` trigger type to create triggers that fire when users attempt to delete a form, after the form is parsed by the Perforce server, but before the form is deleted from the Perforce database.

Example: *An administrator wants to enforce a policy that users are not to delete jobs from the system, but must instead mark such jobs as closed.*

```
#!/bin/sh
echo "Jobs may not be deleted. Please mark jobs as closed instead."
exit 1
```

This `form-delete` *trigger fires on* `job` *forms only. To use the trigger, add the following line to the trigger table:*

```
sample9   form-delete  job  "nodeljob.sh"
```

Whenever a user attempts to delete a job, the request to delete the job is rejected, and the user is shown an error message.

Form-commit triggers

Unlike the other form triggers, the `form-commit` trigger fires after a form is committed to the database. Use these triggers for processes that assume (or require) the successful submission of a form. In the case of job forms, the job's name is not set until after the job has been committed to the database; the `form-commit` trigger is the only way to obtain the name of a new job as part of the process of job creation.

Example: *The following script, when copied to* `newjob.sh`*, shows how to get a job name during the process of job creation, and also reports the status of changelists associated with job fixes.*

```
#!/bin/sh
# newjob.sh - illustrate form-commit trigger
COMMAND=$0
USER=$1
FORM=$2
ACTION=$3
echo $COMMAND: User $USER, formname $FORM, action $ACTION >> log.txt
```

To use the trigger, add the following line to the trigger table:

```
sample10  form-commit  job   "newjob.sh %user% %formname% %action%"
```

Use the `%action%` *variable to distinguish whether or not a change to a job was prompted by a user directly working with a job by means of* `p4 job`*, or indirectly by means of fixing the job within the context of* `p4 fix` *or the* `Jobs:` *field of a changelist.*

The simplest case is the creation of a new job (or a change to an existing job) with the `p4 job` *command; the trigger fires, and the script reports the user, the name of the newly-created (or edited) job. In these cases, the* `%action%` *variable is null.*

The trigger also fires when users add or delete jobs to changelists, and it does so regardless of whether the changed jobs are being manipulated by means of `p4 fix`, `p4 fix -d`, *or by editing the* `Jobs:` *field of the changelist form provided by* `p4 change` *or* `p4 submit` *form). In these cases, the* `%action%` *variable holds the status of the changelist (*`pending` *or* `submitted`*) to which the jobs are being added or deleted.*

Because the `%action%` *variable is not always set, it must be the last argument supplied to any* `form-commit` *trigger script.*

Using triggers for external authentication

To configure Perforce to work with an external authentication manager (such as LDAP or Active Directory), use *authentication triggers* (`auth-check` and `auth-set`). These triggers fire on the `p4 login` and `p4 passwd` commands, respectively.

Authentication triggers differ from changelist and form triggers in that passwords typed by the user as part of the authentication process are supplied to authentication scripts as standard input; never on the command line. (The only arguments passed on the command line are those common to all trigger types, such as `%user%`, `%clientip%`, and so on.)

Warning! Be sure to spell the trigger name correctly when you add the trigger to the trigger table because a misspelling can result in all users being locked out of Perforce.

Be sure to fully test your trigger and trigger table invocation prior to deployment in a production environment.

Contact Perforce Technical Support if you need assistance with restoring access to your server.

The examples in this book are for illustrative purposes only. For a more detailed discussion, including links to sample code for an LDAP environment, see the following Tech Note:

`http://www.perforce.com/perforce/technotes/note074.html`

You must restart the Perforce server after adding an `auth-check` trigger in order for it to take effect. You can, however, change an existing `auth-check` trigger table entry (or trigger script) without restarting the server.

After an `auth-check` trigger is in place and the server restarted, the Perforce `security` counter is ignored; because authentication is now under the control of the trigger script, the server's default mechanism for password strength requirements is redundant.

Auth-check triggers

Triggers of type `auth-check` fire when users run the `p4 login` command. If the script returns `0`, login is successful, and a ticket file is created for the user.

Example: *A trivial authentication-checking script.*

All users must enter the password "secret" before being granted login tickets. Passwords supplied by the user are sent to the script on stdin.

```
#!/bin/bash
# checkpass.sh - a trivial authentication-checking script
# in this trivial example, all users have the same "secret" password
USERNAME=$1
PASSWORD=secret
# read user-supplied password from stdin
read USERPASS
# compare user-supplied password with correct password
if [ "$USERPASS" = $PASSWORD ]
then
    # Success
    exit 0
fi
# Failure
echo checkpass.sh: password $USERPASS for $USERNAME is incorrect
exit 1
```

This `auth-check` *trigger fires whenever users run* `p4 login`*. To use the trigger, add the following line to the trigger table:*

```
sample11  auth-check  auth  "checkpass.sh %user%"
```

Users who enter the "secret" password are granted login tickets.

Auth-set triggers

Triggers of type `auth-set` fire when users run the `p4 passwd` command and successfully validate their old password with an `auth-check` trigger. The process is as follows:

1. A user invokes `p4 passwd`.
2. The Perforce server prompts the user to enter his or her old password.
3. The Perforce server fires an `auth-check` trigger to validate the old password against the external authentication service.
4. The script associated with the `auth-check` trigger runs. If the `auth-check` trigger fails, the process ends immediately: the user is not prompted for a new password, and the `auth-set` trigger never fires.
5. If the `auth-check` trigger succeeds, the Perforce server prompts the user for a new password.
6. The Perforce server fires an `auth-set` trigger and supplies the trigger script with both the old password and the new password on the standard input, separated by a newline.

> **Note** In most cases, users in an external authentication environment will continue to set their passwords without use of Perforce. The `auth-set` trigger type is included mainly for completeness.

Because the Perforce server must validate the user's current password, you must have a properly functioning `auth-check` trigger before attempting to write an `auth-set` trigger.

Example: *A trivial authentication-setting script*

```
#!/bin/bash
# setpass.sh - a trivial authentication-setting script

USERNAME=$1

read OLDPASS
read NEWPASS

echo setpass.sh: $USERNAME attempted to change $OLDPASS to $NEWPASS
```

This `auth-set` *trigger fires after users run* `p4 passwd` *and successfully pass the external authentication required by the* `auth-check` *trigger. To use the trigger, add the following two lines to the trigger table:*

```
sample11  auth-check  auth  "checkpass.sh %user%"
sample12  auth-set    auth  "setpass.sh %user%"
```

This trivial example doesn't actually change any passwords; it merely reports back what the user attempted to do.

Archive triggers for external data sources

The `archive` trigger type is used in conjunction with the `+X` filetype modifier in order to replace the mechanism by which the Perforce Server archives files within the repository. See "Triggers and Windows" on page 127 for platform-specific considerations.

Example: *An archive trigger*

This `archive` *trigger fires when users access files that have the* `+X` *(archive) modifier set.*

```
#!/bin/sh
# archive.sh - illustrate archive trigger

OP=$1
FILE=$2
REV=$3

if [ "$OP" = read ]
then
    cat ${FILE}${REV}
fi

if [ "$OP" = delete ]
then
    rm ${FILE}${REV}
fi

if [ "$OP" = write ]
then
    # Create new file from user's submission via stdin
    while read LINE; do
        echo ${LINE} >> ${FILE}${REV}
    done
    ls -t ${FILE}* |
    {
        read first; read second;
        cmp -s $first $second
        if [ $? -eq 0 ]
        then
            # Files identical, remove file, replace with symlink.
            rm ${FILE}${REV}
            ln -s $second $first
        fi
    }
fi
```

To use the trigger, add the following two lines to the trigger table:

```
arch  archive  arch  "archive.sh %op% %file% %rev%"
```

When the user attempts to submit (write) a file of type `+X`*, if there are no changes between the current revision and the previous revision, the current revision is replaced with a symlink pointing to the previous revision.*

Using multiple triggers

Changelist and form triggers are run in the order in which they appear in the triggers table. If you have multiple triggers of the same type that fire on the same path, each is 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*

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

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

If any trigger fails (for instance, `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.*

To link multiple file specifications to the same trigger (and trigger type), list the trigger multiple times in the trigger table.

Example: *Activating the same trigger for multiple filespecs*

```
Triggers:
  bugcheck change-submit //depot/*.c   "/usr/bin/check4.sh %change%"
  bugcheck change-submit //depot/*.h   "/usr/bin/check4.sh %change%"
  bugcheck change-submit //depot/*.cpp "/usr/bin/check4.sh %change%"
```

In this case, the `bugcheck` *trigger runs on the* `*.c` *files, the* `*.h` *files, and the* `*.cpp` *files.*

Multiple changelist submission triggers of different types that fire on the same path fire in the following order:

1. `change-submit` (fired on changelist submission, before file transmission)
2. `change-content` triggers (after changelist submission and file transmission)
3. `change-commit` triggers (on any automatic changelist renumbering by the server)

Similarly, form triggers of different types are fired in the following order:

1. `form-out` (form generation)
2. `form-in` (changed form is transmitted to the server)
3. `form-save` (validated form is ready for storage in the Perforce database)
4. `form-delete` (validated form is already stored in the Perforce database)

Writing triggers to support multiple Perforce servers

To call the same trigger script from more than one Perforce server, use the `%serverhost%`, `%serverip%`, and `%serverport%` variables to make your trigger script more portable.

For instance, if you have a script that uses hardcoded port numbers and addresses...

```
#!/bin/sh
# Usage: jobcheck.sh changelist
CHANGE=$1
P4CMD="/usr/local/bin/p4 -p 192.168.0.12:1666"
$P4CMD describe -s $1 | grep "Jobs fixed...\n\n\t" > /dev/null
```

...and you call it with the following line in the trigger table...

```
jc1 change-submit //depot/qa/... "jobcheck.sh %change%"
```

...you can improve portability by changing the script as follows...

```
#!/bin/sh
# Usage: jobcheck.sh changelist server:port
CHANGE=$1
P4PORT=$2
P4CMD="/usr/local/bin/p4 -p $P4PORT"
$P4CMD describe -s $1 | grep "Jobs fixed...\n\n\t" > /dev/null
```

...and passing the server-specific data as an argument to the trigger script:

```
jc2 change-submit //depot/qa/... "jobcheck.sh %change% %serverport%"
```

For a complete list of variables that apply for each trigger type, see "Trigger script variables" on page 111.

Triggers and security

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

Triggers and Windows

By default, the Perforce service runs under the Windows local `System` account. The `System` account may have different environmental configurations (including not just Perforce-related variables, but `PATH` settings and file permissions) than the one in which you are using to test or write your trigger.

In general, it is good practice to:

- Wherever possible, use the full path to executables.
- For path names that contain spaces, use the short path name.

  For example, `C:\Program Files\Perforce\p4.exe` is most likely located in `C:\PROGRA~1\Perforce\p4.exe`.
- Login tickets may not be located in the same place as they were during testing; for testing, you can pass in data with `p4 login < input.txt`.
- For troubleshooting, log output to a file. For example:

  ```
  date /t >> trigger.log
  p4 info >> trigger.log
  C:\PROGRA~1\Perforce\p4.exe -p myserver:myport info
  ```

  If the `trigger.log` isn't updated at all, the trigger was not fired. If the first `p4 info` fails, and the second `p4 info` runs, you'll know whether there were differences in environmental settings.
- 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 147.
- On Windows, standard input does not default to binary mode. In text mode, line ending translations are performed on standard input, which is inappropriate for binary files.

  If you are using archive triggers against binary files on a Windows machine, you *must* prevent unwanted line-ending translations by ensuring that standard input is changed to binary mode (`O_BINARY`).

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 Perforce change review daemon (`p4review.py`) is available from the Perforce Supporting Programs page:

```
http://www.perforce.com/perforce/loadsupp.html#daemon
```

The review daemon runs under Python, available at `http://www.python.org/`. Before you run the review daemon, be sure to read and follow the configuration instructions included in the daemon itself.

Users subscribe to files by calling `p4 user`, entering their email addresses in the `Email:` field, and entering any number of file patterns corresponding to files in which they're interested in to the `Reviews:` field.

```
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
```

The change review daemon monitors the files that were included in each newly submitted changelist and emails all users who have subscribed to any files included in a changelist, letting those users know that the files in question have changed.

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 at 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` (see "Perforce's change review daemon" on page 128) 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://public.perforce.com`.

Commands used by daemons

Certain Perforce commands are used almost exclusively by review daemons. These commands are listed in the following table.

Command	Usage
`p4 review -c change#`	For all changelists between *change#* and the latest submitted changelist, this command lists the changelists' numbers, creators, and creators' email addresses. Requires at least `review` access to run.
`p4 reviews -c change# filespec`	Lists all users who have subscribed to review the named files or any files in the specified changelist.
`p4 counter name [value]`	To create a new counter or set the value of an existing counter, you must have `review` access or greater. To display a counter's value, you must have `list` access or greater. If a *value* argument is not included, `p4 counter` returns the value of *name*, or `0` if the counter does not exist. If a *value* argument is included, `p4 counter` sets the value of the *name* to *value*. If the counter *name* does not exist, it is created. ***WARNING:*** The review counters `journal`, `job`, and `change` are used internally by Perforce; ***use of any of these three names as review numbers could corrupt the Perforce database.*** Counters are represented internally as strings. (Prior to release 2008.1, they were stored as signed `int`s)
`p4 counters`	List all counters and their values.
`p4 changes -m 1 -s submitted`	Outputs 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 might also want 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 can 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 might also experience buffering issues introduced by the kernel, because 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 might want to revisit your system configuration to ensure that it is configured for optimal performance.

This 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

In general, Perforce performs well on any server-class hardware platform. 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 bottleneck can be avoided by ensuring that the server doesn't page when it runs 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 1 kilobyte of RAM per file to avoid paging; 10,000 files will require 10 MB 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 three revisions per file, or about 0.5 kilobytes of RAM per file.

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

For most installations, a system with 1.5 kilobytes of RAM per file in the depot suffices.

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 most disk-intensive activity is file check-in, where the Perforce server must write and rename files in

the archive. Server performance depends heavily on the operating system's filesystem implementation, and in particular, on whether directory updates are synchronous. Server performance is also highly dependent upon the capabilities of the underlying hardware's I/O subsystem.

Although Perforce does not recommend any specific hardware configuration or filesystem, Linux servers are generally fastest (owing to Linux's asynchronous directory updating), but they 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.

Performance in systems where database and versioned files are stored on NFS-mounted volumes is typically dependent on the implementation of NFS in question 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 because 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 client programs, (such as `p4`, the Perforce 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 0.5 kilobytes per user per file. (For instance, a system with 10,000 files and 50 users requires 250 MB 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 sufficient disk space for anticipated growth" on page 20.

The `db.have` file holds the list of files opened in client workspaces. This file tends to grow more rapidly 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 client workspace specifications and reducing the scope of client workspace views can help alleviate the problem.

Monitoring disk space usage

To find out how much disk space is currently occupied by files in a depot (or part of a depot), use the `p4 sizes` command with a block size corresponding to that used by your storage solution. For example, the command

```
p4 sizes -a -s -b 512 //depot/...
```

shows the sum (`-s`) of all revisions (`-a`) in `//depot/...` with a block size of 512 bytes.

```
//depot/... 34161 files 277439099 bytes 5429111 blocks
```

Network

Perforce can run over any TCP/IP network. Although we have not yet seen network limitations, the more bandwidth the better.

Perforce uses a TCP/IP 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. Although 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.

CPU

Perforce is based on a client/server architecture. Both the client and server are lightweight in terms of CPU resource consumption; 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. Although 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 can 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.

Windows wildcards

In some cases, `p4` commands on Windows can result in a delayed response if they use unquoted filepatterns with a combination of depot syntax and wildcards, such as:

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

You can prevent the delay by putting double quotes around the file pattern, like this:

```
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 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 might 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 that 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 might 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 on 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 operate on, or at least refer to, all files in the client workspace view.

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

  Because commands without arguments operate on all files in the workspace 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, because the server 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 press 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 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 compression efficiently

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 a Perforce administrator 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 142 and "The temporary client workspace trick" on page 143) 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 81) 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, they also 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*, *maxscanrows*, and *maxlocktime* value. The default for each is `unset`, but a superuser can use `p4 group` to limit it for any given group.

MaxResults prevents the server from using excessive memory by limiting the amount of data buffered during command execution. Users in limited groups are unable to run any commands that buffer more database rows than the group's `MaxResults` limit. (For most sites, `MaxResults` should be larger than the largest number of files anticipated in any one user's individual client workspace.)

Like `MaxResults`, `MaxScanRows` prevents certain user commands from placing excessive demands on the server. (Typically, the number of rows scanned in a single operation is roughly equal to `MaxResults` multiplied by the average number of revisions per file in the depot.)

Finally, `MaxLockTime` is used to prevent certain commands from locking the database for prolonged periods of time. Set `MaxLockTime` to the number of milliseconds for the longest permissible database lock.

To set these limits, fill in the appropriate fields in the `p4 group` form. If a user is listed in multiple groups, the *highest* of the `MaxResults` (or `MaxScanRows`, or `MaxLockTime`) limits (including `unlimited`, but *not* including the default `unset` setting) for those groups is taken as the user's `MaxResults` (or `MaxScanRows`, or `MaxLockTime`) value.

Example: *Effect of setting* `maxresults`, `maxscanrows`, *and* `maxlocktime`.

As an administrator, you want members of the group `rocketdev` *to be limited to operations of 20,000 files or less, that scan no more than 100,000 revisions, and lock database tables for no more than 30 seconds:*

```
Group:        rocketdev
MaxResults:   20000
MaxScanRows:  100000
MaxLockTime:  30000
Timeout:      43200
Subgroups:
Owners:
Users:
        bill
        ruth
        sandy
```

Suppose that Ruth has an unrestricted ("loose") client view. 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, as follows:*

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

Either method enables her to sync her files to her workspace, but without tying up the server to process a single extremely large command.

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

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

If there are fewer 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 does not apply, but the* `MaxScanRows` *limit does.*

Regardless of which limits are in effect, no command she runs will be permitted to lock the database for more than the `MaxLockTime` *of 30000 milliseconds.*

To remove any limits on the number of result lines processed (or database rows scanned, or milliseconds of database locking time) for a particular group, set the `MaxResults` or `MaxScanRows`, or `MaxLockTime` value for that group to `unlimited`.

Because these limitations can make life difficult for your users, do not use them unless you find that certain operations are slowing down your server. Because some Perforce client programs can perform large operations, you should typically set `MaxResults` no smaller than 10,000, set `MaxScanRows` no smaller than 50,000, and `MaxLockTime` to somewhere within the1000-30000 (1-30 second) range.

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

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

from the command line.

MaxResults, MaxScanRows and MaxLockTime for users in multiple groups

As mentioned earlier, if a user is listed in multiple groups, the highest numeric `MaxResults` limit of all the groups a user belongs to is the limit that affects the user.

The default value of `unset` is *not* a numeric limit; if a user is in a group where `MaxResults` is set to `unset`, he or she is still limited by the highest numeric `MaxResults` (or `MaxScanRows` or `MaxLockTime`) setting 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 any of the groups of which the user is a member have their `MaxResults` set to `unlimited`.

Scripting efficiently

The Perforce Command-Line Client, `p4`, supports the scripting of any command that can be run 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 a more efficient approach like this:

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

Instead of an inefficient approach like:

```
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, and then passing the list file to `p4 -x`.

For example, if your script might look something like this:

```
for components in header1 header2 header3
do
    p4 edit ${component}.h
done
```

A more efficient alternative would be:

```
for components in header1 header2 header3
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, you might 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, and then scan the output for the files it needs.

For example, this:

```
p4 files path/...@label | egrep "path/f1.h|path/f2.h|path/f3.h"
```

imposes a lighter load on the Perforce server than either this:

```
p4 files path/f1.h@label path/f1.h@label path/f3.h@label
```

or this:

```
p4 files path/f1.h@label
p4 files path/f2.h@label
p4 files path/f3.h@label
```

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

The temporary client workspace trick

Most Perforce commands can process all the files in the current workspace view with a single command-line argument. By making use of a temporary client workspace with a view that contains only the files on which you want to work, you might be able to reduce the number of commands you have to run, 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 workspace specification that looks like this:

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

Using this workspace specification, you can then run:

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

Using compression efficiently

By default, revisions of files of type `binary` are compressed when stored on the Perforce server.

Some file formats (for example, .GIF and .JPG images, .MPG and .AVI media content, files compressed with `.gz` and `.ZIP` compression) include compression as part of the file format. Attempting to compress such files on the Perforce server results in the consumption of server CPU resources with little or no savings in disk space.

To disable server storage compression for these file types, specify such files as type `binary+F` (binary, stored on the server in full, without compression) either from the command line or from the `p4 typemap` table.

For more about `p4 typemap`, including a sample typemap table, see "Defining filetypes with p4 typemap" on page 48.

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 can become sufficiently unbalanced that performance is negatively affected. The Perforce checkpoint and restore processes (see "Backup and recovery concepts" on page 25) re-create the trees in a balanced manner, and consequently, you might see some increase in server performance following a backup, a removal of the `db.*` files, and the re-creation of the `db.*` files from a checkpoint.

Rebalancing the trees is normally useful only 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, 2000, XP, Vista, and 7.

Using the Perforce installer

The Perforce installer program, `perforce.exe`, gives you the option to install or upgrade the Perforce Server, Perforce Proxy, or the Perforce Command-Line Client.

Separate installers are provided for other Perforce client programs such as P4V.

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 must have Administrator privileges to install Perforce as a service or 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 stops the upgrade, and you are still able to use your old server (or service).

Scripted deployment and unattended installation

The Perforce installer supports scripted installation, enabling you to accelerate a deployment of Perforce across a large number of desktops.

Scripted installations are controlled by a configuration file that comes with the scriptable version of the Perforce installer. You can edit this file to preconfigure Perforce environment variables (such as `P4PORT`) for your environment, to automatically select Perforce client programs in use at your site, and more.

To learn more about how to automate a deployment of Perforce, see Tech Note 68 at:

```
http://kb.perforce.com/?article=68
```

Perforce technical support personnel are available to answer any questions or concerns you have about automating your Perforce deployment.

Windows services vs. Windows servers

To run any task as a Windows *server*, a user account must be logged in, because 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` invokes 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 under the **Control Panel**. You can also stop the Perforce service from the command line with the command:

```
p4 admin stop
```

Starting and stopping the Perforce server

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`

To start a server with settings different from those set by `P4ROOT`, `P4PORT`, `P4LOG`, or `P4JOURNAL`, use `p4d` command-line flags. For example:

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

starts 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`. The `p4d` command-line flags are case-sensitive.

To stop the Perforce server, use the command:

```
p4 admin stop
```

Note If you are running Release 99.1 or earlier, press Ctrl-C in the Command Prompt window, or simply close the window.

Although this method of stopping a Perforce server works in all versions of Perforce, it is not necessarily "clean". With the availability of the `p4 admin stop` command in 99.2, this method 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, when running, runs under this user name, rather than `System`)

Although the Perforce service runs reliably using a network drive as the server root, there is still a marked performance penalty due to increased network traffic and slower file access. Consequently, Perforce recommends that the depot files and Perforce database reside 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 want to host more than one Perforce installation on the same machine (for instance, one for production and one for testing), you can either manually start Perforce servers from the command line, or use the Perforce-supplied utility `svcinst.exe`, to configure additional Perforce services.

Warning! Setting up multiple services to increase the number of users you support without purchasing more user licenses is a violation of the terms of your Perforce End User License Agreement.

Understanding the precedence of environment variables in determining Perforce configuration is useful when configuring multiple Perforce services on the same machine. Before you begin, read and understand "Windows configuration parameter precedence" on page 149.

To set up a second Perforce service:

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. (The `svcinst.exe` utility comes with the Perforce installer, and can be found in your Perforce server root.)
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 is called `Perforce` and its server root directory contains files that are required by any other Perforce services you create on the machine.

Example: *Adding a second Perforce service.*

You want to create a second Perforce service with a root in `C:\p4root2` *and a service name of* `Perforce2`*. The* `svcinst` *executable is in the server root of the first Perforce installation you installed in* `C:\perforce`*.*

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

```
p4d -V
```

(If you are running an older release, you must first download a more recent release from `http://www.perforce.com` *and upgrade your 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 Perforce's `svcinst.exe` *(the service installer) to create the "*`Perforce2`*" service:*

```
svcinst create -n Perforce2 -e c:\p4root2\p4s.exe -a
```

After you create the `Perforce2` *service, 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 Perforce 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 can be set in many different ways. When a Perforce client program (such as `p4` or P4V), 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`)

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
- The **User Variables** tab under the **System Properties** dialog box in the Control Panel

System environment variables can be set with:

- The **System Variables** tab under the **System Properties** dialog box in the Control Panel

Resolving Windows-related instabilities

Many large sites run Perforce servers on Windows without incident. There are also sites in which a 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.

Though not all Perforce failures are caused by OS-level problems, a number of symptoms can 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 service not starting properly.

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 can 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.

Users having trouble with P4EDITOR or P4DIFF

Your Windows users might experience difficulties using the Perforce Command-Line 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, when a Windows command is run (such as a GUI-based editor like `notepad.exe`) from the shell, the shell doesn't always wait for the command to complete before terminating. When this happens, the Perforce client then mistakenly behaves as if 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 error messages 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 use `cmd.exe` only 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 batch file whose contents are the command:

  ```
  start /wait program %1 %2
  ```

  where *`program`* is the name of the editor or diff utility you want to invoke. The `/wait` flag instructs the system to wait for the editor or diff utility to terminate, enabling the Perforce client program to behave properly.

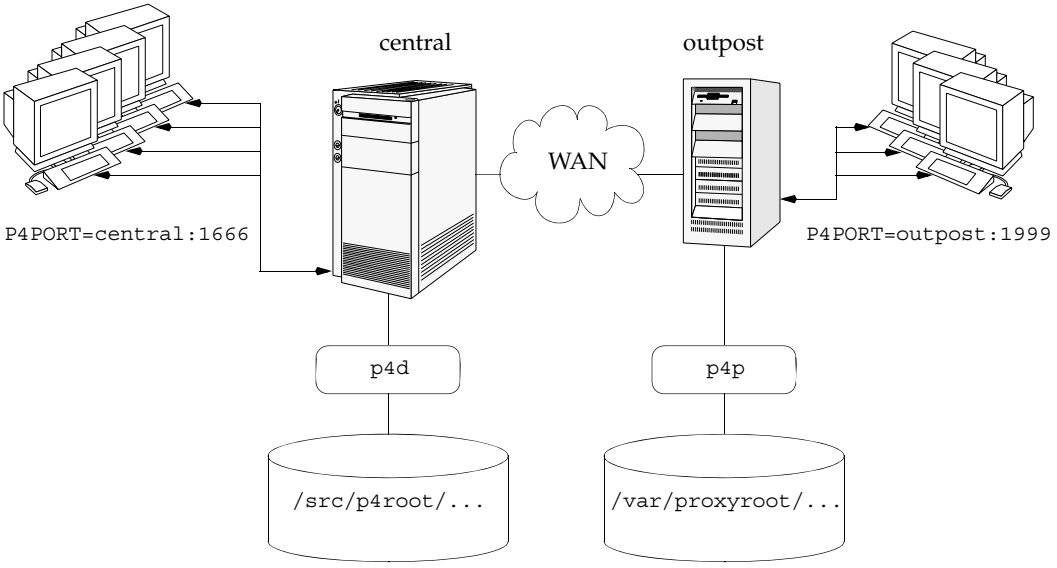
  Some Windows editors (most notably, Wordpad) do not exhibit proper behavior, even when instructed to wait. There is presently no workaround for such programs.

Chapter 9 Perforce Proxy

Perforce is built to handle distributed development in a wide range of network topologies. Where bandwidth to remote sites is limited, P4P, the Perforce Proxy, improves performance by mediating between Perforce clients and servers to cache frequently transmitted file revisions. By intercepting requests for cached file revisions, P4P reduces demand on the Perforce server and network.

To improve performance obtained by multiple Perforce clients accessing a central Perforce server across a WAN, configure P4P on the side of the network close to the clients and configure the clients to access P4P; then configure P4P to access the central Perforce server. (On a LAN, you can also obtain performance improvements by setting up proxies to divert workload from the central server's CPU and disks.)

The following diagram illustrates a typical P4P configuration.



In this configuration, file revisions requested by users at a remote development site are fetched first from a central Perforce server (`p4d` running on `central`) and transferred over a relatively slow WAN. Subsequent requests for that same revision, however, are delivered from the Perforce Proxy, (`p4p` running on `outpost`), over the remote development site's LAN, reducing both network traffic across the WAN and CPU load on the central server.

System requirements

To use Perforce Proxy, you must have:

- A Perforce server at Release 2002.2 or higher
- Sufficient disk space on the proxy host to store a cache of file revisions

Installing P4P

UNIX

To install P4P on UNIX, do the following:

1. Download the `p4p` executable to the machine on which you want to run the proxy.
2. Select a directory on this machine (`P4PCACHE`) in which to cache file revisions.
3. Select a port (`P4PORT`) on which `p4p` will listen for requests from Perforce client programs.
4. Select the target Perforce server (`P4TARGET`) for which this proxy will cache.

Windows

Install P4P from the Windows installer's custom/administrator installation dialog.

Running P4P

To run P4P, invoke the `p4p` executable, configuring it with environment variables or command-line flags. Flags you specify on the command line override environment variable settings.

For example, the following command line starts a proxy that communicates with a central Perforce server located on a host named `central`, listening on port 1666.

```
p4p -p 1999 -t central:1666 -r /var/proxyroot
```

To use the proxy, Perforce client programs connect to P4P on port 1999 on the machine where the proxy runs. P4P file revisions are stored under a directory named `/var/proxyroot`.

Running P4P as a Windows service

To run P4P as a Windows service, either install P4P from the Windows installer, or specify the `-s` flag when you invoke `p4p.exe`, or rename the P4P executable to `p4ps.exe`.

To pass parameters to the P4Proxy service, set the `P4POPTIONS` registry variable using the `p4 set` command. For example, if you normally run the Proxy with the command:

```
p4p -p 1999 -t mainserver:1666
```

you can set the `P4POPTIONS` variable for a Windows service named `Perforce Proxy` by setting the service parameters as follows:

```
p4 set -S "Perforce Proxy" P4POPTIONS="-p 1999 -t mainserver:1666"
```

When you run P4P under the `"Perforce Proxy"` service, the Proxy will listen to port 1999 and communicate with the Perforce Server at `mainserver:1666`.

P4P flags

The following command-line flags specific to the proxy are supported.

Flag	Meaning
`-c`	Do not compress files transmitted from the Perforce server to P4P. (This option reduces CPU load on the central server at the expense of slightly higher bandwidth consumption.)
`-d`	Run as daemon - fork first, then run (UNIX only).
`-f`	Do not fork - run as a single-threaded server (UNIX only).
`-i`	Run for `inetd` (socket on `stdin/stdout` - UNIX only).
`-q`	Run quietly; suppress startup messages.
`-s`	Run as an NT service (Windows only). Running `p4p.exe -s` is equivalent to invoking `p4ps.exe`.
`-e` *`size`*	Cache only those files that are larger than *`size`* bytes. Default is `P4PFSIZE`, or zero (cache all files) if `P4PFSIZE` is not set.

The following general options are supported.

Flag	Meaning
`-h` or `-?`	Display a help message.
`-L` *`logfile`*	Specify the location of the log file. Default is `P4LOG`, or the directory from which `p4p` is started if `P4LOG` is not set.
`-p` *`port`*	Specify the port on which P4P will listen for requests from Perforce client programs. Default is `P4PORT`, or 1666 if `P4PORT` is not set.

Flag	Meaning
`-r root`	Specify the directory where revisions are cached. Default is `P4PCACHE`, or the directory from which `p4p` is started if `P4PCACHE` is not set.
`-t port`	Specify the port of the target Perforce server (that is, the Perforce server for which P4P acts as a proxy). Default is `P4TARGET` or `perforce:1666` if `P4TARGET` is not set.
`-v level`	Specifies server trace level. Debug messages are stored in the proxy server's log file. Debug messages from `p4p` are not passed through to `p4d`, and debug messages from `p4d` are not passed through to instances of `p4p`. Default is `P4DEBUG`, or none if `P4DEBUG` is not set.
`-V`	Display the version of the Perforce Proxy.

Administering P4P

No backups required

You never need to back up the P4P cache directory.

If necessary, P4P reconstructs the cache based on Perforce server metadata.

Stopping P4P

P4P is effectively stateless; to stop P4P under UNIX, `kill` the `p4p` process with `SIGTERM` or `SIGKILL`. Under Windows, click **End Process** in the **Task Manager**.

Managing disk space consumption

P4P caches file revisions in its cache directory. These revisions accumulate until you delete them. P4P does not delete its cached files or otherwise manage its consumption of disk space.

Warning! If you do not delete cached files, you will eventually run out of disk space.

To recover disk space, remove files under the proxy's root. It is safe to delete the proxy's cached files while the proxy is running.

Determining if your Perforce client is using the proxy

If your Perforce client program is using the proxy, the proxy's version information appears in the output of `p4 info`.

For example, if a Perforce server is running on `central:1666` and you direct your Perforce client to a Perforce Proxy running on `outpost:1999`, the output of `p4 info` resembles the following:

```
$ export P4PORT=outpost:1999
$ p4 info
User name: p4adm
Client name: admin-temp
Client host: remotesite22
Client root: /home/p4adm/tmp
Current directory: /home/p4adm/tmp
Client address: 192.168.0.123:55768
Server address: central:1666
Server root: /usr/depot/p4d
Server date: 2008/06/28 15:03:05 -0700 PDT
Server uptime: 752:41:23
Server version: P4D/FREEBSD4/2008.1/156375 (2008/05/25)
Proxy version: P4P/SOLARIS26/2008.1/156884 (2008/05/25)
Server license: P4 Admin <p4adm> 20 users (expires 2009/01/01)
Server license-ip: 10.0.0.2
```

P4P and protections

To apply the IP address of a Perforce Proxy user's workstation against the protections table, prepend the string `proxy-` to the workstation's IP address.

For instance, consider an organization with a remote development site with workstations on a subnet of `192.168.10.0/24`. The organization also has a central office where local development takes place; the central office exists on the `10.0.0.0/8` subnet. A Perforce server resides on the `10.0.0.0/8` subnet, and a Perforce Proxy resides on the `192.168.10.0/24` subnet. Users at the remote site belong to the group `remotedev`, and occasionally visit the central office.

To ensure that members of the `remotedev` group use the proxy while working at the remote site, but do not use the proxy when visiting the local site, add the following lines to your protections table:

```
list    group    remotedev     192.168.10.0/24       -//...
write   group    remotedev     proxy-192.168.10.0/24  //...
list    group    remotedev     proxy-10.0.0.0/8      -//...
write   group    remotedev     10.0.0.0/8             //...
```

The first line denies `list` access to all users in the `remotedev` group if they attempt to access Perforce without using the proxy from their workstations in the `192.168.10.0/24` subnet. The second line grants write access to all users in `remotedev` if they are using a Perforce Proxy server and are working from the `192.168.10.0/24` subnet. Users of workstations at the remote site must use the proxy.

Similarly, the third and fourth lines deny `list` access to `remotedev` users when they attempt to use the proxy from workstations on the central office's subnet (`10.0.0.0/8`), but grant write access to `remotedev` users who access the Perforce server directly from workstations on the central office's subnet. When visiting the local site, users from the `remotedev` group must access the Perforce server directly.

Determining if specific files are being delivered from the proxy

Use the `-Zproxyverbose` flag with `p4` to display messages indicating whether file revisions are coming from the proxy (`p4p`) or the central server (`p4d`).

For instance:

```
$ p4 -Zproxyverbose sync noncached.txt
//depot/main/noncached.txt - refreshing /home/p4adm/tmp/noncached.txt
$ p4 -Zproxyverbose sync cached.txt
//depot/main/cached.txt - refreshing /home/p4adm/tmp/cached.txt
File /home/p4adm/tmp/cached.txt delivered from proxy server
```

Maximizing performance improvement

Reducing server CPU usage by disabling file compression

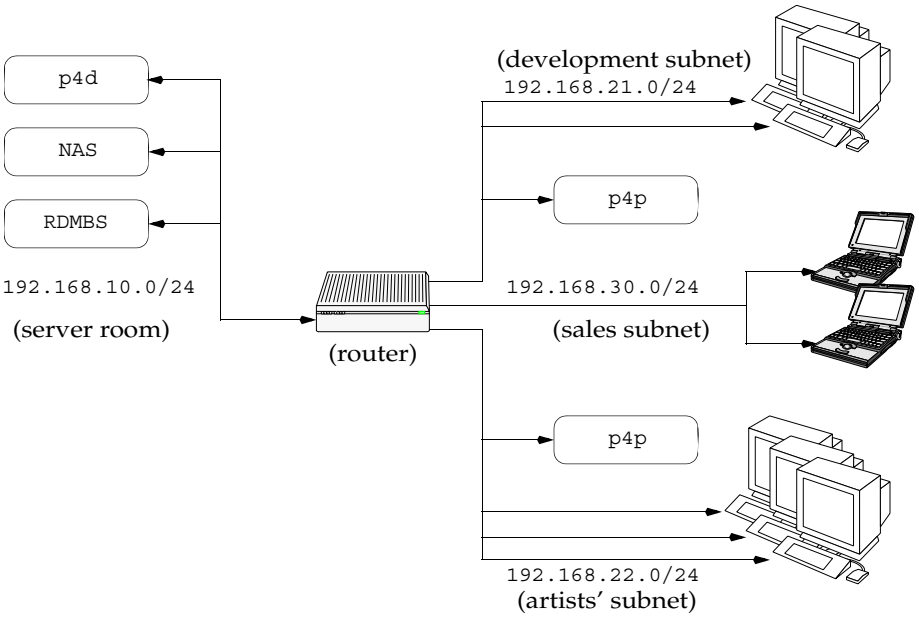
By default, P4P compresses communication with the central Perforce server, imposing additional overhead on the server.

To disable compression, specify the `-c` option when you invoke `p4p`. This option is particularly effective if you have excess network and disk capacity and are storing large numbers of binary file revisions in the depot, because the proxy (rather than the server) decompresses the binary files from its cache before sending them to Perforce clients.

Network topologies versus P4P

If network bandwidth on the same subnet as the central Perforce server is nearly saturated, deploying proxy servers on the same subnet will not likely result in a performance improvement. Instead, deploy the proxy servers on the other side of a router so that the traffic from the clients to the proxy server is isolated to a subnet separate from the subnet containing the central Perforce server.

For example:



Deploying an additional proxy server on a subnet when network bandwidth on the subnet is nearly saturated will not likely result in a performance improvement. Instead, split the subnet into multiple subnets and deploy a proxy server in each resulting subnet.

In the illustrated configuration, a server room houses a company's Perforce server (`p4d`), a network storage device (NAS), and a database server (RDBMS). The server room's network segment is saturated by heavy loads placed on it by a sales force constantly querying a database for live updates, and by developers and graphic artists frequently accessing large files through the Perforce server.

Deploying two instances of Perforce Proxy (one on the developers' subnet, and one on the graphic artists' subnet) enables all three groups to benefit from improved performance due to decreased use on the server room's network segment.

Preloading the cache directory for optimal initial performance

P4P stores file revisions only when one of its clients submits a new revision to the depot or requests an existing revision from the depot. That is, file revisions are not prefetched. Performance gains from P4P occur only after file revisions are cached.

After starting P4P, you can effectively prefetch the cache directory by creating a dedicated client workspace and syncing it to the head revision. All other clients that subsequently connect to the proxy immediately obtain the performance improvements provided by P4P. For example, a development site located in Asia with a P4P server targeting a Perforce server in North America can preload its cache directory by using an automated job that runs a `p4 sync` against the entire Perforce depot after most work at the North American site has been completed, but before its own developers arrive for work.

By default, `p4 sync` writes files to the client workspace. If you have a dedicated client workspace that you use to prefetch files for the proxy, however, this step is redundant. If the client machine has slower I/O performance than the machine running the Perforce Proxy, it can also be time-consuming.

To preload the proxy's cache without the redundant step of also writing the files to the client workspace, use the `-Zproxyload` option when syncing. For example:

```
$ export P4CLIENT=prefetch
$ p4 sync //depot/main/written.txt
//depot/main/written.txt - refreshing /home/prefetch/main/written.txt
$ p4 -Zproxyload sync //depot/main/nonwritten.txt
//depot/main/nonwritten.txt - file(s) up-to-date.
```

Both files are now cached, but `nonwritten.txt` is never written to the the `prefetch` client workspace. When prefetching the entire depot, the time savings can be considerable.

Distributing disk space consumption

P4P stores revisions as if there were only one depot tree. If this approach stores too much file data onto one filesystem, you can use symbolic links to spread the revisions across multiple filesystems.

For instance, if the P4P cache root is `/disk1/proxy`, and the Perforce server it supports has two depots named `//depot` and `//released`, you can split data across disks, storing `//depot` on `disk1` and `//released` on `disk2` as follows:

```
mkdir /disk2/proxy/released
cd /disk1/proxy
ln -s /disk2/proxy/released released
```

The symbolic link means that when P4P attempts to cache files in the `//released` depot to `/disk1/proxy/released`, the files are stored on `/disk2/proxy/released`.

Chapter 10 Perforce Replication

What is Replication?

Replication is the duplication of server data from one Perforce Server to another Perforce Server, ideally in real time. Replication is used to reduce server load, for example, by using the replicated data for reporting tasks, and for disaster recovery, to minimize downtime and data loss.

Perforce's `p4 replicate` utility replicates server metadata (that is, the information contained in the `db.*` files) unidirectionally from one server to another (the replica server). To achieve a fully functional replica server, you can configure the replica server to read the source server's versioned files (the `,v` files that contain the deltas that are produced when new versions are submitted), or use third-party or native OS functionality to mirror the versioned content to the replica server's root.

Replica servers are intended to be used for read-only purposes. If you require read/write access to a remote server, use the Perforce Proxy.

Why Replicate?

You can use replication to:

- Reduce load and downtime on the primary server

  Long-running queries and reports, builds, and checkpoints can be run against a replica server, reducing lock contention. For reports and checkpoints, only metadata needs to be replicated. For builds, ensure that the build processes have read access (but not write access) to the master server's versioned files.

- Provide warm standby servers

  You can use `p4 replicate` to maintain a replica server as an up-to-date warm standby system, to be used if the master server fails. These replica servers require that both server metadata and versioned files are replicated. The `p4 replicate` command replicates metadata only. To mirror the master server's versioned files, use tools such as rsync.

How does replication work?

Metadata replication works by using the same data as Perforce's backup and restore features.

The `p4 replicate` command maintains transactional integrity while reading transactions out of an originating server and copying them into a replica server's database.

You run `p4 replicate` on the machine that will host the replica Perforce Server. `p4 replicate` fetches journal records from the master Perforce Server, and provides these records to a subprocess that restores the records into a database.

Finally, you run a `p4d` for the replica Perforce Server that points to the database being restored. Users may connect to the replica server as they would connect to the master server.

System requirements

- The master and replica servers must be revision 2009.2 or higher.
- The replica server must have a license file. (Any license file will do; it does not have to be the same license file as used on the master server.)
- The master and replica servers must have the same time zone setting.

Windows	On Windows, the time zone setting is system-wide. On UNIX, the time zone setting is controlled by the `TZ` environment variable at the time the replica server is started.

Warnings

- Bidirectional replication is not supported, because it can corrupt the versioned files on both the master and replica servers.

Limits and restrictions

- `p4 replicate` does not read compressed journals. Therefore, the master server must not compress rotated journals until the replica server has fetched all journal records from older journals.
- `p4 replicate` only replicates metadata. If your application requires the replication of (or access to) archive files, these archive files need to be accessible to the replica Perforce Server. Use a network mounted file system or utilities such as rsync or SAN replication.

- Client workspaces on the master and replica servers must be mutually exclusive. Users must be certain that their `P4PORT`, `P4CLIENT`, and other settings are configured such that files from the replica server are not inadvertently synced to client workspaces used with the master server, nor vice versa.
- Users must not submit changes to the replica server. To prevent submission of changes, set permissions on the replica archive files to read-only (relative to the userid that owns the replica `p4d` process.) Other mechanisms, such as the use of trigger scripts on both the master and the replica server, written such that they would always succeed on the master server and fail on the replica server, could be used to enforce this requirement.

Replication commands

Replica servers are supplied with metadata from a master server by means of a `p4 replicate` command. The `p4 replicate` command polls the master server for new journal entries, and outputs them either on standard output, or pipes the records to a subprocess, also supplied on the command line.

To start a replica server, you must first run `p4 replicate`:

```
p4 replicate [replicate-flags] command [command-flags]
```

and then start the replica server:

```
p4d -r replicaroot -p replicaport
```

In most cases, the *`command`* supplied to `p4 replicate` will be a variation of `p4d -jr`, to read in the journal records from the master server into a set of database files used by the replica server.

Warning! The replica server's root must not be the same as the master server's root.

Always specify the replica server's root by means of a command line flag, or from within a script. The command line flags override any `P4ROOT` variable setting, reducing the risk that you will inadvertently start a replica server that points to your master server's `db.*` files.

For further protection, it is recommended to run your replica server as a different user than the one used to run your master server. The userid that owns the replica server's process should not have write privileges to any of the directories managed by the master server.

p4 replicate flags

A typical invocation of `p4 replicate` looks like this:

```
p4 replicate -s statefile -i interval [-k] [-x] [-J prefix] command
```

Use `-s` to specify the state file. The state file is a one-line text file that determines where subsequent invocations of `p4 replicate` will start reading data. The format is *`journalno/byteoffset`*, where *`journalno`* is the number of the most recent journal, and *`byteoffset`* is the number of bytes to skip before reading. If you do not specify a byte offset, `p4 replicate` starts reading at the start of the specified journal file. If no statefile exists, replication commences with the first available journal, and a statefile is created for you.

Use `-i` to specify the polling *`interval`* in seconds. The default is two seconds. To disable polling (that is, to check once for updated journal entries and then exit), specify an *`interval`* of 0.

Use `-J` *`prefix`* to specify a filename prefix for the journal, such as that used with `p4d -jc` *`prefix`*.

By default, `p4 replicate` shuts down the pipe to the command process between polling intervals; in most cases, you will want to use -k to keep the pipe to the *`command`* subprocess open. This is the most common use case.

> **Note** If you are using `-k`, you must use the `-jrc` (see "p4 replicate commands" on page 163) option for consistency checking. Conversely, if you are *not* using `-jrc`, do not use `-k` to keep the connection open.

By default, `p4 replicate` continues to poll the master server until it is stopped by the user. Use `-x` to have `p4 replicate` exit when journal rotation is detected. This option is typically used in offline checkpointing configurations.

For a complete list of all the flags usable with `p4 replicate`, see the *Command Reference* entry for `p4 replicate`.

p4 replicate commands

A typical *command* supplied to `p4 replicate` looks like this:

```
p4d -r replicaroot -f -b 1 -jrc -
```

The invocation of the `p4d` command makes use of three new (as of release 2009.2) flags to `p4d`:

The `-jrc` flag instructs `p4d` to check for consistency when reading journal records. Batches of journal records are read in increasing size until `p4d` processes a marker which indicates that all transactions are complete. After the marker is processed, the affected database tables are locked, the changes are applied, and the tables are unlocked. Because the server always remains in a state of transactional integrity, it is possible for other users to use the replica server while the journal transactions are being applied from the master server.

The `-b 1` flag refers to bunching journal records, sorting them, and removing duplicates before updating the database. The default is 5000 records per update, but in the case of replication, serial processing of journal records (a bunch size of 1) is required; hence, each line is read individually.

The `-f` flag supplied to `p4d` in conjunction with the `-jr` flag forces `p4d` to ignore failures to delete records. This flag is required for certain replication configurations because some tables on the replica server (depending on use cases) will differ from those on the master server.

Replication Examples

Offloading Reporting and Checkpointing Tasks

The basic replication configuration consists of a master server and a replica server. The replica server has a replica of the master server's metadata, but none of the versioned files. This configuration is useful for offloading server-intensive report generation, and for performing offline checkpoints.

1. Create a server root directory for the replica server.

   ```
   replica$ mkdir /usr/replica/root
   ```

2. Checkpoint the master server:

   ```
   master$ p4d -r /usr/master/root -jc
   ```

   You could also use `p4 -p master:1666 admin checkpoint`.

3. When the checkpoint is complete, copy the checkpoint file (`checkpoint.`*nnn*) to the replica server's server root and note the checkpoint sequence number (*nnn*).

4. On the replica server, recover the checkpoint to the replica server's root directory:

   ```
   replica$ p4d -r /usr/replica/root -jr checkpoint.nnn
   ```

   This saves time by creating an initial set of `db.*` files for the replica server to use; from this point forward, transfer of data from the master will be performed by `p4 replicate`.

5. Install a license file in the `P4ROOT` directory on the replica server.

6. `p4 replicate` keeps track of its state (the most recent checkpoint sequence number read, and a byte offset for subsequent runs) in a statefile.

   The first time you run `p4 replicate`, you must set the checkpoint sequence number in the statefile as follows:

   ```
   replica$ echo nnn > state
   ```

7. Start replication by issuing the following command:

   ```
   replica$ p4 -p master:1666 -u super replicate -s state -i 10 -k p4d
   -r /usr/replica/root -f -b 1 -jrc -
   ```

   The specified user requires `super` access on the master server (in this case, `master:1666`).

   The default polling interval is 2 seconds. (In this example, we have used `-i 10` to specify a polling interval of 10 seconds.)

8. The replica server's metadata is now being updated by `p4 replicate`; you can now start the replica server itself:

   ```
   replica$ p4d -p 6661 -r /usr/replica/root
   ```

   Users should now be able to connect to the replica server on `replica:6661` and run basic reporting commands (`p4 jobs`, `p4 filelog`, and so on) against it.

Using replica servers

The replica server can be stopped, and checkpoints can be performed against it, just as they would be with the master server. The advantage is that the checkpointing process can now take place without any downtime on the master server.

Users connect to replica servers by setting `P4PORT` as they would with any other server.

Commands which require access to versioned file data (`p4 sync`, for example) will fail, because this basic configuration replicates only the metadata, but does not have access to the versioned files.

Configuring for Continuous Builds

To support continuous builds, a replica server requires read-only access to the versioned files that reside on the master server.

Because the depot table (`db.depot`) is part of the replicated metadata, the replica server must access the versioned files through paths that are identical to those on the master server. If the master server's depot table has archives specified with full paths (that is, if local depots on the master server were specified with absolute paths, rather than paths relative to the master server's root directory), the same paths must exist on the replica server. If the master server's local depots were specified with relative paths, the versioned files *can* be located anywhere on the replica server by using links from the replica server's root directory.

Warning! Replication is unidirectional.

To prevent users from submitting changes to the replica server, run the replica `p4d` as a user that does not have write access to the master server's archive files.

Providing a Warm Standby Server

To support warm standby servers, a replica server requires an up-to-date copy of both the master server's metadata and its versioned files. To replicate the master server's metadata, use `p4 replicate`.

To replicate the versioned files from the master server, use native operating system or third-party utilities. Ensure that the replicated versioned files can remain synchronized with the metadata (check the polling interval used by `p4 replicate` to replicate transactions being applied to the master server's `db.*` files). The `%changeroot%` trigger variable may be of use; it contains the top-level directory of files affected by a changelist.

Disaster recovery and failover strategies are complex and tend to be site-specific. Perforce Consultants are available to assist organizations in the planning and deployment of disaster recovery and failover strategies.

```
http://perforce.com/perforce/services/consulting.html
```

Next Steps

For more information about replication, see the Perforce Knowledge Base article:

```
http://kb.perforce.com/?article=1099
```

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 involving checkpointing and journaling.

On UNIX and Mac OS 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 following 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 or journaling flags, `p4d` exits with a nonzero error code if any error occurs.

If you attempt to use Zeroconf and mDNS (multicast DNS) is not available, only the service registration fails. The error is logged in the server log, and the server continues with startup.

Options

Flag	Meaning
`-0`	Register the server as a zero-configuration (Zeroconf) service. Overrides `P4ZEROCONF` setting.
`-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)
`-s`	Run `p4d.exe` as an NT service (equivalent to running `p4s.exe`).
`-xi`	Irreversibly reconfigure the Perforce server (and its metadata) to operate in Unicode mode. Do not use this flag unless you know you require Unicode mode. See the *Release Notes* and *Internationalization Notes* for details.
`-xu`	Run database upgrades and exit.
`-c` *`command`*	Lock database tables, run *`command`*, unlock the tables, and exit.
`-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.
`-jrc` *`file`*	Journal-restore with integrity-checking. Because this option locks the database, this option is intended only for use by replica servers started with the `p4 replicate` command.
`-b` *`bunch`* `-jr` *`file`*	Read *`bunch`* lines of journal records, sorting and removing duplicates before updating the database. The default is 5000, but can be set to 1 to force serial processing. This combination of flags is intended for use with by replica servers started with the `p4 replicate` command.
`-f -jr` *`file`*	Ignore failures to delete records; this meaning of `-f` applies only when `-jr` is present. This combination of flags is intended for use with by replica servers started with the `p4 replicate` command.
`-z`	Compress (in `gzip` format) checkpoints and journals.
`-h, -?`	Print help message.

Flag	Meaning
`-V`	Print server version.
`-A` *`auditlog`*	Specify an audit log file. Overrides `P4AUDIT` setting. Default is null.
`-Id` *`description`*	A service description (optional) that will be advertised as a zero-configuration (Zeroconf) service. Use with `-0`. Overrides `P4DESCRIPTION` setting.
`-In` *`name`*	A service name (required) that will be advertised as a zero-configuration (Zeroconf) service. Use with `-0`. Overrides `P4NAME` setting.
`-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` *`subsystem=level`*	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 want 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 can vary, depending on whether 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 of the filesystem containing `P4ROOT`, the journal is likely to remain accessible.

- The database upgrade flag (`-xu`) can require considerable disk space. See the *Release Notes* for details when upgrading.
- Zeroconf service registration requires support for mDNS (multicast DNS) and DNS-SD (DNS-based service discovery) on the server machine, such as Bonjour (for OS X), Bonjour for Windows (for Windows), or Avahi (for Linux or UNIX).

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 -J /p4d/jfile -jc`
	or
	`P4JOURNAL=/p4d/jfile ; export P4JOURNAL`
	`p4d -jc`

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`
To advertise availability as a zero-configuration service	`p4d -0 -In "Artists\' storage" -Id "Use this server for art assets"`

Index