Perforce 2002.2 User's Guide

December 2002

Table of Contents

Preface

About This Manual

This is the *Perforce 2002.2 User's Guide*. It teaches the use of Perforce's Comnand-Line Client. Other Perforce clients such as P4Win (the Perforce Windows Client) are not discussed here. If you'd like documentation on other Perforce client programs, please see our documentation pages, available from our web site at `http://www.perforce.com`.

Although you can use this guide as a reference manual, we intend it primarily as a guide/tutorial on using Perforce. The full syntax of most of the Perforce commands is explicitly not provided here; in particular, only a subset of the available flags are mentioned. For a complete guide to Perforce, please see the *Perforce Command Reference*, or the on-line help system. If you will be using Perforce on any operating system other than UNIX, please consult the Perforce platform notes for that OS.

Chapters 2 through 4 of this manual comprise our *Getting Started* guide. Newcomers to Perforce should start there, and move to subsequent chapters as needed.

Administering Perforce?

If you're administering a Perforce server, you'll need the *Perforce System Administrator's Guide*, which contains all the system administration material formerly found in this manual. If you're installing Perforce, the *Perforce System Administrator's Guide* is also the place to start.

Please Give Us Feedback

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

Chapter 1 Product Overview

Perforce facilitates the sharing of files among multiple users. It is a software configuration management tool, but software configuration management (SCM) has been defined in many different ways, depending on who's giving the definition. SCM has been described as providing version control, file sharing, release management, defect tracking, build management, and a few other things. It's worth looking at exactly what Perforce does and doesn't do:

- Perforce offers *version control*: multiple revisions of the same file are stored and older revisions are always accessible.
- Perforce provides facilities for *concurrent development*; multiple users can edit their own copies of the same file.
- Perforce supports *distributed development*; users can work with files stored on a central server or with files replicated on a proxy server.
- Some *release management* facilities are offered; Perforce can track the file revisions that are part of a particular release.
- Bugs and system improvement requests can be tracked from entry to fix; this capability is known as *defect tracking* or *change management*.
- Perforce supplies some *lifecycle management* functionality; files can be kept in release branches, development branches, or in any sort of needed file set.
- *Change review* functionality is provided by Perforce; this functionality allows users to be notified by email when particular files are changed.
- Although a build management tool is not built into Perforce, we do offer a companion open source product called `Jam`. The `Jam` tool and Perforce meet at the file system; source files managed by Perforce are easily built by `Jam`.

Although Perforce was built to manage source files, it can manage any sort of on-line documents. Perforce can be used to store revisions of a manual, to manage Web pages, or to store old versions of operating system administration files.

Perforce Server and Perforce Client Programs

Perforce has a client/server architecture, in which users at other computers are connected to one central machine, the *server*. Each user works on a client machine; at their command, a Perforce client program transfers files to and from the Perforce server. The client programs communicate with the server using TCP/IP.

The Perforce clients can be distributed around a local area network, wide area network, dialup network, or any combination of these topologies. Perforce clients can also reside on the same host as the server.

The following programs do the bulk of Perforce's work:

- The Perforce Server (`p4d`) runs on the Perforce server machine. It manages the shared file repository, and keeps track of users, workspaces, and other Perforce metadata.

  The `p4d` program must be run on a UNIX or Windows machine.

- Perforce client programs (for instance, `p4`) run on Perforce client machines. Client programs send users' requests to the Perforce Server (`p4d`) for processing, and communicate with `p4d` using TCP/IP.

  Perforce client programs can be run on many platforms, including UNIX, Linux, Windows, VMS, Macintosh, BeOS, and NeXT hosts.

This manual assumes that you or your system administrator have already installed both `p4` and `p4d`. You'll find installation instructions in the *Perforce System Administrator's Guide*, also available at our Web site.

Moving files between the clients and the server

Users create, edit, and delete files in their own directories on client machines; these directories are called *client workspaces*. Perforce commands are used to move files to and from a shared file repository on the server known as the *depot*. Perforce users can retrieve files from the depot into their own client workspaces, where they can be read, edited, and resubmitted to the depot for other users to access. When a new *revision* of a file is stored in the depot, the old revisions are kept and are still accessible.

Files that have been edited within a client workspace are sent to the depot using a *changelist*, which is a list of files and instructions that tell the depot what to do with those files. For example, one file might have been changed in the client workspace, another added, and another deleted. These file changes can be sent to the depot in a single changelist, which is processed *atomically*: either all the changes are made to the depot at once, or none of them are. This approach allows users to simultaneously update all files related to a bug fix or a new feature.

Each client workspace has its own *client view*, which determines which files in the depot can be accessed by that client workspace. One client workspace might be able to access all the files in the depot, while another client workspace might access only a single file. The Perforce Server is responsible for tracking the state of the client workspace; Perforce knows which files a client workspace has, where they are, and which files have write permission turned on.

For basic information about using Perforce, see Chapter 3, Perforce Basics: Quick Start and Chapter 4, Perforce Basics: The Details.

File conflicts

When two users edit the same file, their changes can conflict. For example, suppose two users copy the same file from the depot into their workspaces, and each edits his copy of the file in different ways. The first user sends his version of the file back to the depot, and then the second user tries to do the same thing. If Perforce were to unquestioningly accept the second user's file into the depot, the first user's changes would not be included in the latest revision of the file (known as the *head revision*).

When a file conflict is detected, Perforce allows the user experiencing the conflict to perform a *resolve* of the conflicting files. The resolve process allows the user to decide what needs to be done: should his file overwrite the other user's? Should his own file be thrown away? Or should the two conflicting files be merged into one? At the user's request, Perforce will perform a *three-way merge* between the two conflicting files and the single file that both were based on. This process generates a *merge* file from the conflicting files, which contains all the changes from both conflicting versions. This file can be edited and then submitted to the depot.

To learn how to resolve file conflicts, see Chapter 5, Perforce Basics: Resolving File Conflicts.

Labeling groups of files

It is often useful to mark a particular set of file revisions for later access. For example, the release engineers might want to keep a list of all the file revisions that comprise a particular release of their program. This list of files can be assigned a name, such as `release2.0.1`; this name is a *label* for the user-determined list of files. At any subsequent time, the label can be used to copy its revisions into a client workspace.

For details about labels, see Chapter 8, Labels.

Branching files

Thus far, it has been assumed that all changes to files happen linearly, but this is not always the case. Suppose that one source file needs to evolve in two separate directions; perhaps one set of upcoming changes will allow the program to run under VMS, and

another set will make it a Mac program. Clearly, two separately-evolving copies of the same files are necessary.

Perforce's *Inter-File Branching*™ mechanism allows any set of files to be copied within the depot. By default, the new file set, or *codeline*, evolves separately from the original files, but changes in either codeline can be propagated to the other.

For details about branching, see Chapter 9, Branching.

Job tracking

A *Job* is a generic term for a plain-text description of some change that needs to be made to the source code. A job might be a bug description, like "the system crashes when I press return", or it might be a system improvement request, like "please make the program run faster."

Whereas a job represents work that is intended to be performed, a *changelist* represents work actually done. Perforce's job tracking mechanism allows jobs to be linked to the changelists that implement the work requested by the job. A job can later be looked up to determine if and when it was fixed, what files were modified to implement the fix, who fixed it, and whether the fix has been propagated to other codelines. The fields contained in your system's jobs can be defined by the Perforce system administrator.

Perforce's job tracking mechanism does not implement all the functionality that is normally supplied by full-scale defect tracking systems. Its simple functionality can be used as is, or it can be integrated with third-party job tracking systems through P4DTI - Perforce Defect Tracking and Integration.

To read more about jobs, please see Chapter 10, Job Tracking.

Change notification

Perforce's *change review* mechanism allows users to receive email notifying them when particular files have been updated in the depot. The files for which a particular user receives notification are determined by that user. Change review is implemented by an external program, or *daemon*, which can be customized.

Perforce can be made to run external scripts whenever changelists are submitted. These scripts, called *triggers*, allow changelists to be validated before they're submitted to the depot.

To learn how to set up the change review daemon, integrate Perforce with third-party defect tracking systems, or write your own daemons, consult the *Perforce System Administrator's Guide.*

Protections

Perforce provides a protection scheme to prevent unauthorized or inadvertent access to the depot. The protection mechanism determines exactly which Perforce commands are allowed to be run by any particular client.

Permissions can be granted or denied based on users' usernames and IP addresses, or can be granted or denied to entire groups of users. Because Perforce usernames are easily changed, protections at the user level provide safety, not security. Protections at the IP address level are as secure as the host itself.

We discuss protections in the *Perforce System Administrator's Guide.*

Other Perforce Clients

The Perforce Command-Line Client (`p4`) is not the only Perforce client program. Other Perforce client programs, including P4Win, the Perforce Windows Client, may be downloaded from the Perforce web site.

P4Win

The Perforce Windows Client provides a native Microsoft Windows user interface for all SCM tasks. Using the familiar Windows® Explorer look and feel, it shows you your work in progress at a glance and lets you point, click, drag, and drop your way through Perforce tasks.

For more about P4Win, see the product page at:

```
http://www.perforce.com/perforce/products/p4win.html
```

P4Web

The Perforce web client turns most any Web browser into a complete SCM tool. P4Web will work with a Perforce Server at Release 99.2 or newer, and runs on Unix, Macintosh, and Windows.

For more about P4Web, see the product page at:

```
http://www.perforce.com/perforce/products/p4web.html
```

Merge Tools

Interactive merge tools allow you to display the differences between file versions, simplifying the process of resolving conflicts that result from parallel or concurrent development efforts. Merge tools often use color-coding to highlight differences and some even include the option to automatically merge non-conflicting changes.

Perforce offers full support for both parallel and concurrent development environments. In situations where concurrent file check-out is not desirable, Perforce can be configured to restrict this capability to specific file types or file locations (for instance, management of digital assets in environments where concurrent development is not encouraged).

P4 resolve

Perforce's "`p4 resolve`" command includes built-in merge capability for the console environment.

P4WinMerge

P4WinMerge is Perforce's graphical three-way merge and conflict resolution tool for Windows. P4WinMerge uses the "3-pane" approach to display and edit files during the merge process.

P4WinMerge is a stand-alone Windows application; it does not require a Perforce Server when used by itself. However, when invoked from within a Perforce client program like the Perforce Command-Line Client, P4Win, or P4Web, a Perforce Server is necessary.

For more about P4WinMerge, see:

`http://www.perforce.com/perforce/products/p4winmerge.html`

Other merge utilities

Perforce is easily integrated with third-party merge tools and diff utilities. Users need only change an environment variable (such as `P4MERGE` or `P4DIFF`) to point to their merge tool of choice.

For more about using third-party merge tools with Perforce, see:

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

Defect Tracking Systems

Perforce provides a number of options for defect tracking. In addition to providing basic built-in defect tracking, Perforce is integrated with several leading defect tracking systems. Activity performed by Perforce users can be automatically sent to your defect tracking system. Conversely, issues and status entered into your defect tracking system can be accessed by Perforce users.

Perforce jobs

Perforce's built-in defect tracking and reporting features are available to all Perforce users.

P4DTI integrations with third-party defect trackers

Although Perforce provides built-in defect tracking, some companies prefer to use the defect tracking system they've already got in place, or want to install a different defect tracker for use with Perforce.

Perforce Defect Tracking Integration (P4DTI) is an open source project specifically designed to integrate Perforce with other defect tracking systems by replicating Perforce jobs and changelist numbers to their equivalents in the other system.

P4DTI connects your defect tracking system to Perforce, so that you don't have to switch between your defect tracker and SCM tool and enter duplicate information about your work. P4DTI also links changes made in Perforce with defect tracker issues, making it easy to find out why a change was made, find the work that was done to resolve an issue, or generate reports relating issues to files or codelines.

Activity in your Perforce depot - enhancements, bug fixes, propagation of changes into release branches, and so forth - can be automatically entered into your defect tracking system by P4DTI. Conversely, issues and status entered into your defect tracking system - bug reports, change orders, work assignments, and so on, can be converted automatically to Perforce metadata for access by Perforce users. With P4DTI, you can integrate Perforce with any third-party defect tracking or process management software.

P4DTI uses Perforce's built-in "jobs" feature to mirror data in defect tracking systems. While Perforce jobs can be used without additional software for straightforward issue tracking, P4DTI lets you take advantage of third-party user interfaces, reporting tools, databases, and workflow rules to manage complex processes.

P4DTI runs on Unix and Windows. It can be used with a Perforce Server on any platform at Release 2000.2 or newer.

For more about using third-party defect tracking systems with Perforce, including a list of defect tracking systems for which P4DTI integrations have already been built, see:

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

Plug-Ins, reporting and tool integrations

IDE Plug-ins

Perforce IDE Plug-ins allow developers to work with Perforce from within integrated development environments (IDEs) such as Microsoft Developer Studio, Borland JBuilder, and Metrowerks CodeWarrior.

For more about Perforce IDE Plug-ins, see:

```
http://www.perforce.com/perforce/products/plugins-ide.html
```

P4Report and P4SQL

The Perforce Reporting System (P4Report) offers query and reporting capability for Perforce depots. P4Report also includes the Perforce SQL Command-Line Client (P4SQL). P4SQL can be used to execute SQL statements either interactively or using scripts.

Based on P4ODBC, the Perforce ODBC Data Source, P4Report can be used by ODBC-compliant reporting tools including Crystal Reports®, Microsoft® Access® and Excel®. P4Report can also be integrated with some defect tracking systems.

For more about P4Report and P4SQL, see:

```
http://www.perforce.com/perforce/products/p4report.html
```

P4OFC

The Perforce Plug-in for Microsoft Office (P4OFC) adds a "Perforce" menu to Microsoft Word, Microsoft Excel, and Microsoft Powerpoint. This menu provide easy access to common Perforce SCM commands, so that users never have to leave familiar office applications to work with documents under Perforce control.

For more about P4OFC, see:

```
http://www.perforce.com/perforce/products/plugins-ofc.html
```

Chapter 2 Connecting to the Perforce Server

Perforce uses a client/server architecture. Files are created and edited by users on their own client hosts; these files are transferred to and from a shared file repository located on a Perforce server. Every running Perforce system uses a single server and can have many clients.

As mentioned earlier, two programs do the bulk of Perforce's work:

- The `p4d` program runs on the Perforce server. It manages the shared file repository, and keeps track of users, clients, protections, and other Perforce metadata.
- The `p4` program runs on each Perforce client. It sends the users' requests to the `p4d` server program for processing, and communicates with `p4d` via TCP/IP.

Each Perforce client program needs to know the address and port of the Perforce server with which it communicates. This address is stored in the `P4PORT` environment variable.

Before you begin

This chapter assumes that your system administrator has already set up a Perforce server (`p4d`) for you, and that it is already up and running.

If this is not the case (for instance, if you're installing Perforce from scratch), you'll also have to install the Perforce server before continuing. See the appendix, "Installing Perforce" on page 137, for information on how to install the server.

The information in the appendix is intended to help you install a server for evaluation purposes. If you're installing a production server, or are planning on extensive testing of your evaluation server, we strongly encourage you to read the full installation instructions in the *Perforce System Administrator's Guide*.

Setting up your environment to use Perforce

A Perforce client program needs to know two things in order to talk to a Perforce server:

- the name of the host on which `p4d` is running, and
- the port on which `p4d` is listening

These are set via a single environment variable, P4PORT.

Note | See "Setting and viewing environment variables" on page 144 for information about how to set environment variables for most operating systems and shells.

If your site is already using Perforce, it's possible that your system administrator has already set P4PORT for you; if not, you'll need to set it yourself.

Either way, after setting P4PORT to point to your server, you should verify your client's connection to the Perforce server with the p4 info command.

Telling Perforce clients where the server is

To use Perforce, you'll need to know the name of the host where p4d is located, and the number of the TCP/IP port on which it's listening.

- If you've just installed the Perforce server yourself, you already know this, having configured the server on a specific host to listen to a specific port.
- If you're connecting to an existing Perforce installation, you'll have to ask your system administrator for the host and port of the Perforce server.

Once you've obtained the host and port information, set your P4PORT environment variable to *host*:*portNum*, where *host* is the name of the host on which p4d is running, and *portNum* is the port to which it is listening. For example:

If the server is running on...	and is listening to port...	set P4PORT to:
dogs	3435	dogs:3435
x.com	1818	x.com:1818

The definition of P4PORT can be shortened if the Perforce client is running on the same host as the server. In this case, only the port number need be provided to p4. If p4d is running on a host named or aliased perforce, listening on port 1666, the definition of P4PORT for the p4 client can be dispensed with altogether. For example:

If the server is running on...	and is listening to port...	set P4PORT to:
<same host as the p4 client>	1543	1543
perforce	1666	*<no value needed>*

When P4PORT has been set, you should re-verify the connection with p4 info, as described below. Once this has been done, Perforce is ready to use.

Verifying the connection to the Perforce server

To verify the connection, type `p4 info` at the command line. If the `P4PORT` environment variable is correctly set, you'll see something like this:

```
User name: edk
Client name: wrkstn12
Client host: wrkstn12
Client unknown.
Current directory: /usr/edk
Client address: 192.168.0.123:1818
Server address: p4server:1818
Server root: /usr/depot/p4d
Server date: 2000/07/28 12:11:47 -0700 PDT
Server version: P4D/FREEBSD/2000.1/16375 (2000/07/25)
Server license: P4 Admin <p4adm> 20 users on unix (expires 2001/01/01)
```

The `Server address:` field shows which Perforce server to which the client has connected; it displays the host and port number on which `p4d` is listening.

In the above example, everything is fine. If, however, you receive a variant of this message:

```
Perforce client error:
        Connect to server failed; check $P4PORT.
        TCP connect to perforce:1666 failed.
        perforce: host unknown.
```

then `P4PORT` has not been correctly set. If the value you see in the third line of the error message is `perforce:1666` (as above), then `P4PORT` has not been set at all; if the value is anything else, `P4PORT` has been incorrectly set. In either case, you'll need to set the value of `P4PORT`.

Windows On Windows platforms, registry variables are preferred over environment variables, and you can set these with command `p4 set`.

Chapter 3 Perforce Basics: Quick Start

This chapter teaches basic Perforce usage. You'll learn how to set up a workspace, populate it with files from the common file repository (the *depot*), edit these files and submit the changes back to the repository, back out of any unwanted changes, and the basic Perforce reporting commands.

This chapter gives a broad overview of these concepts and commands; for details, see Chapter 4, Perforce Basics: The Details.

Underlying concepts

The basic ideas behind Perforce are quite simple: files are created, edited, and deleted in the user's own directories, which are called *client workspaces*. Perforce commands are used to move files to and from a shared file repository known as the *depot*. Perforce users can retrieve files from the depot into their own client workspaces, where they can be read, edited, and resubmitted to the depot for other users to access. When a new revision of a file is stored in the depot, the old revisions are kept and remain accessible.

Perforce was written to be as unobtrusive as possible, so that very few changes to your normal work habits are required. Files are still created in your own directories with your tool of choice; Perforce commands supplement your normal work actions instead of replacing them.

Perforce commands are always entered in the form `p4 command [arguments]`.

File configurations used in the examples

This manual makes extensive use of examples based on the source code set for a program called Elm. The Elm examples used in this manual are set up as follows:

- A single depot is used to store the Elm files, and perhaps other projects as well. The elm files will be shared by storing them under an `elm` subdirectory within the depot.
- Each user will store his or her client workspace Elm files in a different subdirectory. The two users we'll be following most closely, Ed and Lisa, will work with their Elm files in the following locations:

User	Username	Client Workspace Name	Top of own Elm File Tree
Ed	`edk`	`eds_elm`	`/usr/edk/elm`
Lisa	`lisag`	`lisas_ws`	`/usr/lisag/docs`

Setting up a client workspace

To move files between a client workspace and the depot, the Perforce server requires two pieces of information:

- A name that uniquely identifies the client workspace, and
- The top-level directory of this workspace.

Naming the client workspace

To name your client workspace, or to use a different workspace, set the environment variable `P4CLIENT` to the name of the client workspace

Example: *Naming the client workspace*

Ed is working on the code for Elm. He wants to refer to the collection of files he's working on by the name `eds_elm`*. In the Korn or Bourne shells, he'd type:*

```
$ P4CLIENT=eds_elm ; export P4CLIENT
```

Each operating system or shell has its own method of defining environment variables. See the "Environment Variables" section of the *Perforce Command Reference* for details.

Describing the client workspace to the Perforce server

Once the client workspace has been named, it must be identified and described to the Perforce server with the `p4 client` command. Typing `p4 client` brings up the client definition form in a standard text editor; once the form is filled in and the editor exited, the Perforce server is able to move files between the depot and the client workspace.

Note Many `p4` commands, including `p4 client`, display a form for editing in a standard text editor. You can define the editor your client uses through the `P4EDITOR` environment variable.

The `p4 client` form has a number of fields; the two most important are the `Root` and `View`. The meanings of these fields are as follows:

Field	Meaning
`Root:`	Identifies the top subdirectory of the client workspace. This should be the lowest-level directory that includes all the files and directories that you'll be working with.
`View:`	Describes which files and directories in the depot are available to the client workspace, and where the files in the depot will be located within the client workspace.

Example: *Setting the client root and the client view:*

Ed is working with his Elm files in a setting as described above. He's set the environment variable `P4CLIENT` *to* `eds_elm`*. Now he types* `p4 client` *from his home directory, and sees the following form:*

```
Client: eds_elm
Owner:  ed
Description:
        Created by ed.
Root:   /usr/edk
Options:        nomodtime noclobber
View:
        //depot/...    //eds_elm/...
```

With these default settings, all files in Ed's home directory of `/usr/edk` *(including files unrelated to Ed's work) would be mapped to the depot, and all files in the depot would be mapped to Ed's home directory, likely cluttering it with files Ed has no interest in working with.*

Ed would like to keep all Elm-related material in a subdirectory in his home directory; `/usr/edk/elm`*, and he would like this directory to contain only files in the* `elm_proj` *portion of the depot. He therefore changes the values in the* `Root:` *and* `View:` *fields as follows:*

```
Client: eds_elm
Owner:  ed
Description:
        Created by ed.
Root:   /usr/edk/elm
Options:        nomodtime noclobber
View:
        //depot/elm_proj/...  //eds_elm/...
```

This specifies that `/usr/edk/elm` *is the top level directory of Ed's client workspace, and that the files under this workspace directory are to be mapped to the depot's* `elm_proj` *subtree.*

When Ed is done, he exits from the editor, and the `p4 client` *command saves his changes.*

Warning! To use Perforce properly, it is crucial to understand how views work. Views are explained in more detail at the start of the next chapter.

Windows In Windows environments, a single client workspace may span multiple drives by setting the client root to null and including the drive letter in the client view. Please see the platform notes on the Perforce web site for further details.

The read-only `Client:` field contains the string stored in the `P4CLIENT` environment variable. The `Description:` can be filled with anything at all; it's a place where you can enter text that describes the contents of this client workspace. The `View:` describes the relationship between files in the depot and files in the client workspace.

Creating a client specification has no immediate visible effect; no files are created when a client specification is created or edited. The client specification simply indicates where files will be located when subsequent Perforce commands are used.

You'll also use `p4 client` to change existing client specifications. This is described in "Perforce Basics: The Details" on page 37.

Copying depot files into your workspace

Use `p4 sync` to retrieve files from the depot into a client workspace.

Note If you're setting up a brand new depot, `p4 sync` won't do anything, since there are no files in the depot to copy to the client workspace yet.

If this is the case, start by copying files from your client workspace to the depot with `p4 add`, as described in "Adding files to the depot" on page 29.

Example: *Copying files from the depot to a client workspace.*

Lisa has been assigned to fix bugs in Ed's code. She creates a directory called `elm_ws` *within her own directory, and sets up a client workspace; now she wants to copy all the existing elm files from the depot into her workspace.*

```
$ cd ~/elm_ws
$ p4 sync
//depot/elm_proj/doc/elmdoc.0#2 - added as /usr/lisag/elm_ws/doc/elmdoc.0
//depot/elm_proj/doc/elmdoc.1#2 - added as /usr/lisag/elm_ws/doc/elmdoc.1
<etc.>
```

Once the command completes, the most recent revisions of all the files in the depot that are mapped through her client workspace view will be available in her workspace.

The `p4 sync` command maps depot files through the client view, compares the result against the current client contents, and then adds, updates, or deletes files in the client workspace as needed to bring the client contents in sync with the depot. `p4 sync` can take filenames as parameters, with or without wildcards, to limit the files it retrieves.

If a file exists within a particular subdirectory in the depot, but that directory does not yet exist in the client workspace, the directory is created within the client workspace at sync time.

The job of `p4 sync` is to match the state of the client workspace to that of the depot; thus, if a file has been deleted from the depot, `p4 sync` deletes it from the client workspace.

Updating the depot with files from your workspace

Any file in a client workspace can be added to, updated in, or deleted from the depot. This is accomplished in two steps:

1. Perforce is told the new state of client workspace files with the commands `p4 add` *`filenames`*, `p4 edit` *`filenames`*, or `p4 delete` *`filenames`*. When these commands are given, the corresponding files are listed in a Perforce changelist, which is a list of files and operations on those files to be performed in the depot.
2. The operations are performed on the files in the changelist when the `p4 submit` command is given.

> **Note** This chapter discusses only the *default changelist*, which is automatically maintained by Perforce. Changelists can also be created by the user; for a full discussion, see Chapter 7, Changelists.

The commands `p4 add`, `p4 edit`, and `p4 delete` do not immediately add, edit, or delete files in the depot. Instead, the affected file and the corresponding operation are listed in the *default changelist*, and the files in the depot are affected only when this changelist is submitted to the depot with `p4 submit`. This allows a set of files to be updated in the depot all at once: when the changelist is submitted, either all of the files in the changelist are affected, or none of them are.

When a file has been opened with `p4 add`, `p4 edit`, or `p4 delete`, but the corresponding changelist has not yet been submitted in the depot, the file is said to be *open* in the client workspace.

Adding files to the depot

To add a file or files to the depot, type `p4 add` *`filename(s)`*. The `p4 add` command opens the file(s) for `add` and lists them in the default changelist, but they won't be added to the depot until the `p4 submit` command is given.

Example: *Adding files to a changelist.*

Ed is writing a help manual for Elm. The files are named `elmdoc.0` *through* `elmdoc.3`*, and they're sitting in the* `doc` *subdirectory of his client workspace root. He wants to add these files to the depot.*

```
$ cd ~/elm/doc
$ p4 add elmdoc.*
//depot/elm_proj/doc/elmdoc.0#1 - opened for add
//depot/elm_proj/doc/elmdoc.1#1 - opened for add
//depot/elm_proj/doc/elmdoc.2#1 - opened for add
//depot/elm_proj/doc/elmdoc.3#1 - opened for add
```

At this point, the files he wants to add to the depot have been added to his default changelist. However, the files are not actually stored in the depot until the `p4 submit` command is given.

Example: *Submitting a changelist to the depot.*

Ed is ready to submit his added files to the depot. He types `p4 submit` *and sees the following form in a standard text editor:*

```
Change: new
Client: edk
User:   edk
Status: new
Description:
        <enter description here>
Files:
        //depot/elm_proj/doc/elmdoc.0  # add
        //depot/elm_proj/doc/elmdoc.1  # add
        //depot/elm_proj/doc/elmdoc.2  # add
        //depot/elm_proj/doc/elmdoc.3  # add
```

Ed changes the contents of the `Description:` *field to describe what these file updates do. When he's done, he quits from the editor, and the new files are added to the depot.*

The `Description:` field contents must be changed, or the depot update won't be accepted. Lines can be deleted from the `Files:` field; any files deleted from this list will carry over to the next default changelist, and will appear again the next time `p4 submit` is performed.

Adding more than one file at once

Multiple file arguments can be provided on the command line.

Example: *Using multiple file arguments on a single command line.*

Ed wants to add all of his Elm library, documentation, and header files to the depot.

```
$ cd ~
$ p4 add elm/lib/* elm/hdrs/* elm/doc/*
//depot/elm_proj/lib/Makefile.SH#1 - opened for add
//depot/elm_proj/lib/add_site.c#1 - opened for add
//depot/elm_proj/lib/addrmchusr.c#1 - opened for add
<etc.>
```

After `p4 add` *is finished, Ed then does a* `p4 submit`.

If the directory containing a new file does not exist in the depot, it is automatically created within the depot at *submit* time.

The operating system's write permission on submitted files is turned off in the client workspace when `p4 submit` is performed. This helps ensure that file editing is done with Perforce's knowledge. The write permissions are turned back on by `p4 edit`, which is described below.

You might have noticed in the example above that the filenames are displayed as *`filename`*`#1`. Perforce always displays filenames with a `#`*`N`* suffix; the `#`*`N`* indicates that this is the *`N`*th revision of this file. Revision numbers are always assigned sequentially.

Warning! If a submit fails, the default changelist will be assigned a number, and you'll need to submit that changelist in a slightly different way.

Please see Chapter 5, Perforce Basics: Resolving File Conflicts for instructions on resolving file conflicts.

Populating empty depots

In Perforce, there's no difference between adding files to an empty depot and adding files to a depot that already contains other files. For this reason, populate new, empty depots by adding files from a client workspace with `p4 add`, as described above.

Editing files in the depot

To open a file for `edit`, use `p4 edit`. This has two effects:

- The file(s) write permissions are turned on in the client workspace, and
- The file(s) to be edited are added to the default changelist.

Since the files must have their write permission turned back on before they can be edited, you must give the `p4 edit` command before attempting to edit the file.

To save the new file revision in the depot, use `p4 submit`, as described above.

Note Before a file can be opened for `edit`, it must already have been added to the depot with `p4 add`, or copied into the client workspace from the depot with `p4 sync`.

Example: *Opening a file for* `edit`*:*

Ed wants to make changes to his `elmdoc.3` *file. He opens the file for* `edit`.

```
$ cd ~/elm
$ p4 edit doc/elmdoc.3
//depot/elm_proj/doc/elmdoc.3#1 - opened for edit
```

He then edits the file with any text editor. When he's finished, he submits the file to the depot with `p4 submit`*, as described above.*

Deleting files from the depot

Files are deleted from the depot in a way similar to the way in which they are added and edited.

The `p4 delete` command opens the file for `delete` in the default changelist, and then `p4 submit` is used to delete the file from the depot. The `p4 delete` command also deletes the file from the client workspace; this occurs when the `p4 delete` command is given. The deletion of the file in the depot occurs only after the changelist with the delete operation is submitted.

Example: *Deleting a file from the depot.*

Ed's file `doc/elmdoc.3` *is no longer needed. He deletes it from both his client workspace and from the depot as follows:*

```
$ cd ~/elm/doc
$ p4 delete elmdoc.3
//depot/elm_proj/doc/elmdoc.3#1 - opened for delete
```

The file is deleted from the client workspace immediately, but it is not deleted from the depot until he gives the `p4 submit` *command.*

Once the changelist is submitted, it appears as if the file has been deleted from the depot; however, old file revisions are never actually removed. This makes it possible to read older revisions of "deleted" files back into the client workspace.

Submitting with multiple operations

Multiple files can be included in any changelist. Submitting the changelist to the depot works atomically: either all the files are updated in the depot, or none of them are. (In Perforce's terminology, this is called an *atomic change transaction*). Changelists can be used to keep files together that have a common purpose.

Example: *Adding, updating, and deleting files in a single submit:*

Ed is writing the portion of Elm that is responsible for multiple folders (multiple mailboxes). He has a new source file `src/newmbox.c`*, and he needs to edit the header file* `hdrs/s_elm.h` *and the* `doc/elmdoc` *files. He adds the new file and prepares to edit the existing files:*

```
$ cd ~
$ p4 add elm/src/newmbox.c
//depot/elm_proj/src/newmbox.c#1 - opened for add
<etc.>
$ p4 edit elm/hdrs/s_elm.h doc/elmdoc.*
//depot/elm_proj/hdrs/s_elm.h#1 - opened for edit
//depot/elm_proj/doc/elmdoc.0#1 - opened for edit
//depot/elm_proj/doc/elmdoc.1#1 - opened for edit
//depot/elm_proj/doc/elmdoc.2#2 - opened for edit
```

He edits the existing files and then performs a `p4 submit` *of the default changelist:*

```
Change: new
Client: eds_elm
User:   edk
Status: new
Description:
        Changes to Elm's "mult. mailbox" functionality
Files:
        //depot/elm_proj/doc/elmdoc.0  # edit
        //depot/elm_proj/doc/elmdoc.1  # edit
        //depot/elm_proj/doc/elmdoc.2  # edit
        //depot/elm_proj/hdrs/s_elm.h    # edit
        //depot/elm_proj/src/newmbox.c   # add
```

All of his changes supporting multiple mailboxes are grouped together in a single changelist; when Ed quits from the editor, either all of these files are updated in the depot, or, if the submission fails for any reason, none of them are.

Files can be deleted from the `Files:` field; these files are moved into the next default changelist, and appear again the next time `p4 submit` is performed.

Backing out: reverting files to their unopened states

Any file opened for `add`, `edit`, or `delete` can be removed from its changelist with `p4 revert`. This command reverts the file in the client workspace back to its unopened state, and any local modifications to the file are lost.

Example: *Reverting a file back to the last synced version.*

Ed wants to edit a set of files in his src directory: `leavembox.c`, `limit.c`, *and* `signals.c`. *He opens the files for* `edit`*:*

```
$ cd ~elm/src
$ p4 edit leavembox.c limit.c signals.c
//depot/elm_proj/src/leavembox.c#2 - opened for edit
//depot/elm_proj/src/limit.c#2 - opened for edit
//depot/elm_proj/src/signals.c#1 - opened for edit
```

and then realizes that `signals.c` *is not one of the files he will be working on, and that he didn't mean to open it. He can revert* `signals.c` *to its unopened state with* `p4 revert`*:*

```
$ p4 revert signals.c
//depot/elm_proj/src/signals.c#1 - was edit, reverted
```

If `p4 revert` is used on a file that had been opened with `p4 delete`, it will appear back in the client workspace immediately. If `p4 add` was used to open the file, `p4 revert` removes it from the changelist, but leaves the client workspace file intact.

If the reverted file was originally opened with `p4 edit`, the last synced version will be written back to the client workspace, overwriting the newly-edited version of the file. To reduce the risk of overwriting changes by accident, you may want to preview a revert by using `p4 revert -n` before running `p4 revert`. The `-n` option reports what files would be reverted by `p4 revert` without actually reverting the files.

Basic reporting commands

Perforce provides some 20+ reporting commands. Each chapter in this manual ends with a description of the reporting commands relevant to the chapter topic. All the reporting commands are discussed in greater detail in Chapter 11, Reporting and Data Mining.

The most basic reporting commands are `p4 help` and `p4 info`.

Command	Meaning
`p4 help commands`	Lists all Perforce commands with a brief description of each.
`p4 help` *`command`*	For any *`command`* provided, gives detailed help about that command. For example, `p4 help sync` provides detailed information about the `p4 sync` command.

Command	Meaning
`p4 help usage`	Describes command-line flags common to all Perforce commands.
`p4 help views`	Gives a discussion of Perforce view syntax
`p4 help`	Describes all the arguments that can be given to `p4 help`.
`p4 info`	Reports information about the current Perforce system: the server address, client root directory, client name, user name, Perforce version, and a few other tidbits.

Two other reporting commands are used quite often:

Command	Meaning
`p4 have`	Lists all file revisions that the Perforce server knows you have in the client workspace.
`p4 sync -n`	Reports what files would be updated in the client workspace by `p4 sync` without actually performing the sync operation.

Chapter 4 Perforce Basics: The Details

This chapter covers the Perforce rules in detail. The topics discussed include views, mapping depots to client workspaces, Perforce wildcards, rules for referring to older file revisions, file types, and form syntax. For a brief overview of Perforce, refer to Chapter 3, Perforce Basics: Quick Start.

Description of the Client Workspace

A Perforce client workspace is a collection of source files managed by Perforce on a host. Each collection is given a name which identifies the client workspace to the Perforce server. By default, the name is simply the host's name, but this can be overridden by the environment variable `P4CLIENT`. There can be more than one Perforce client workspace on a client host.

All files within a Perforce client workspace share a common root directory, called the *client root*. In the degenerate case, the client root can be the host's root, but in practice the client root is the lowest level directory under which the managed source files sit.

Perforce manages the files in a client workspace in three direct ways:

- It creates, updates, or deletes files as required in the workspace when the user requests Perforce to synchronize the client workspace with the depot,
- It turns on write permission when the user requests to edit a file, and
- It turns off write permission and submits updated versions back to the depot when the user has finished editing the file and submits his or her changes.

The entire Perforce client workspace state is tracked by the Perforce server. The server knows what files a client workspace has, where they are, and which files have write permission turned on.

Perforce's management of a client workspace requires a certain amount of cooperation from the user. Because client files are ordinary files with `write` permission turned off, willful users can circumvent the system by turning on write permission, directly deleting or renaming files, or otherwise modifying the file tree supposedly under Perforce's control.

Perforce counters this with two measures: first, Perforce has explicit commands to verify that the client workspace state is in accord with the server's recording of that state; second, Perforce tries to make using Perforce at least as easy as circumventing it. For example: to make a temporary modification to a file, it is easier to use Perforce than it is to copy and restore the file manually.

Files not managed by Perforce may also be under a client's root, but are ignored by Perforce. For example, Perforce may manage the source files in a client workspace, while the workspace also holds compiled objects, libraries, and executables, as well as a developer's temporary files.

In addition to accessing the client files, the `p4 client` program sometimes creates temporary files on the client host. Otherwise, Perforce neither creates nor uses any files on the client host.

Wildcards

Perforce uses three wildcards for pattern matching. Any number and combination of these can be used in a single string:

Wildcard	Meaning
`*`	Matches anything except slashes; matches only within a single directory.
`...`	Matches anything including slashes; matches across multiple directories.
`%d`	Used for parametric substitution in views. See "Changing the order of filename substrings" on page 43 for a full explanation.

The "`...`" wildcard is passed by the `p4 client` program to the Perforce server, where it is expanded to match the corresponding files known to `p4d`. The `*` wildcard is expanded locally by the OS shell before the `p4` command is sent to the server, and the files that match the wildcard are passed as multiple arguments to the `p4` command. To have Perforce match the `*` wildcard against the contents of the depot, it must be escaped, usually with quotes or a backslash. Most command shells don't interfere with the other two wildcards.

Wildcards and "p4 add"

The "`...`" wildcard can't be used with the `p4 add` command. The "`...`" wildcard is expanded by the Perforce server, and since the server doesn't know what files are being added (after all, they're not in the depot yet), it can't expand that wildcard. The `*` wildcard may be used with `p4 add`, as it is expanded by the local OS shell, not by the `p4d` server.

Mapping the Depot to the Client Workspace

Just as a client name is simply an alias for a particular directory on the client machine, a depot name is an alias for a directory on the Perforce server. The relationship between files in the depot and files in the client workspace is described in the *client view*, and it is set with the `p4 client` command.

When you type `p4 client`, Perforce displays a variation of the following form:

```
Client: eds_elm
Owner:  edk
Description:
         Created by ed.
Root:   /usr/edk/elm
Options:        nomodtime noclobber
View:
        //depot/...    //eds_elm/...
```

The contents of the `View:` field determine where client files get stored in the depot, and where depot files are copied to in the client.

Note The `p4 client` form has more fields than are described here. For a full discussion, please see the *Perforce Command Reference*.

Multiple depots

By default, there is a single depot in each Perforce server, and the name of the depot is `depot`. The Perforce system administrator can create multiple depots on the same Perforce server.

If your system administrator has created multiple depots on your server, the default client view will look something like this:

```
View:
        //depot/...       //eds_elm/depot/...
        //user_depot/...  //eds_elm/user_depot/...
        //projects/...    //eds_elm/projects/...
```

The *Perforce System Administrator's Guide* explains how to create multiple depots.

Using views

Views consist of multiple lines, or *mappings*, and each mapping has two parts. The left-hand side specifies one or more files within the depot, and has the form:

```
//depotname/file_specification
```

The right-hand side of each mapping describes one or more files within the client workspace, and has the form:

```
//clientname/file_specification
```

The left-hand side of a client view mapping is called the *depot side*, and the right-hand side is the *client side*.

The default view in the example above is quite simple: it maps the entire depot to the entire client workspace. Views can contain multiple mappings, and can be much more complex, but all client views, no matter how elaborate, perform the same two functions:

- *Determine which files in the depot can be seen by a client workspace.*

  This is determined by the sum of the depot sides of the mappings within a view. A view might allow the client workspace to retrieve every file in the depot, or only those files within two directories, or only a single file.

- *Construct a one-to-one mapping between files in the depot and files in the client workspace.*

  Each mapping within a view describes a subset of the complete mapping. The one-to-one mapping might be straightforward; for example, the client workspace file tree might be identical to a portion of the depot's file tree. Or it can be oblique; for example, a file might have one name in the depot and another in the client workspace, or be moved to an entirely different directory in the client workspace. No matter how the files are named, there is always a one-to-one mapping.

To determine the exact location of any client file on the host machine, substitute the value of the `p4 client` form's `Root:` field for the client name on the client side of the mapping. For example, if the `p4 client` form's `Root:` field for the client `eds_elm` is set to `/usr/edk/elm`, then the file `//eds_elm/doc/elmdoc.1` will be found on the client host in `/usr/edk/elm/doc/elmdoc.1`.

Windows On Windows machines, the Perforce client must be specified in a slightly different way if it is to span multiple drives. For details, please see the platform notes at `http://www.perforce.com/perforce/technical.html`

Wildcards in views

Any wildcard used on the depot side of a mapping must be matched with an identical wildcard in the mapping's client side. Any string matched by the wildcard is identical on both sides.

In the client view

```
//depot/elm_proj/...      //eds_elm/...
```

the single mapping contains Perforce's "`...`" wildcard, which matches everything including slashes. The result is that any file in the `eds_elm` client workspace will be

mapped to the same location within the depot's `elm_proj` file tree. For example, the file `//depot/elm_proj/nls/gencat/README` is mapped to the client workspace file `//eds_elm/nls/gencat/README`.

To properly specify file trees, use the "`...`" wildcard after a trailing slash. (If you specify only `//depot/elm_proj...`, then the resulting view also includes files and directories such as `//depot/elm_project_coredumps`, which is probably not what you intended.)

Types of mappings

By changing the `View:` field, it is possible to map only part of a depot to a client workspace. It is even possible to map files within the same depot directory to different client workspace directories, or to have files named differently in the depot and the client workspace. This section discusses Perforce's mapping methods.

Direct client-to-depot views

The default view in the form presented by `p4 client` maps the entire client workspace tree into an identical directory tree in the depot. For example, the default view

```
//depot/... //eds_elm/...
```

indicates that any file in the directory tree under the client `eds_elm` will be stored in the identical subdirectory in the depot. This view is usually considered to be overkill, as most users only need to see a subset of the files in the depot.

Mapping the full client to only part of the depot

Usually, you'll only want to see part of the depot. Change the left-hand side of the `View:` field to point to only the relevant portion of the depot.

Example: *Mapping part of the depot to the client workspace.*

Bettie is rewriting the documentation for Elm, which is found in the depot within its `doc` *subdirectory. Her client is named* `elm_docs`*, and her client root is* `/usr/bes/docs`*; she types* `p4 client` *and sets the* `View:` *field as follows:*

```
//depot/elm_proj/doc/... //elm_docs/...
```

The files in `//depot/elm_proj/doc` *are mapped to* `/usr/bes/docs`*. Files not beneath the* `//depot/elm_proj/doc` *directory no longer appear in Bettie's workspace.*

Mapping files in the depot to different parts of the client

Views can consist of multiple mappings, which are used to map portions of the depot file tree to different parts of the client file tree. If there is a conflict in the mappings, later mappings have precedence over the earlier ones.

Example: *Multiple mappings in a single client view.*

The `elm_proj` *subdirectory of the depot contains a directory called* `doc`*, which has all of the Elm documents. Included in this directory are four files named* `elmdoc.0` *through* `elmdoc.3`*. Mike wants to separate these four files from the other documentation files in his client workspace, which is called* `mike_elm`*.*

To do this, he creates a new directory in his client workspace called `help`*, which is located at the same level as his* `doc` *directory. The four* `elmdoc` *files will go here, so he fills in the* `View:` *field of the* `p4 client` *form as follows:*

```
View:
        //depot/...                         //mike_elm/...
        //depot/elm_proj/doc/elmdoc.*       //mike_elm/help/elmdoc.*
```

Any file whose name starts with `elmdoc` *within the depot's* `doc` *subdirectory is caught by the later mapping and appears in Mike's workspace's* `help` *directory; all other files are caught by the first mapping and appear in their normal location. Conversely, any files beginning with* `elmdoc` *within Mike's client workspace* `help` *subdirectory are mapped to the* `doc` *subdirectory of the depot.*

Note Whenever you map two sections of the depot to different parts of the client workspace, some depot and client files will remain unmapped. See "Two mappings can conflict and fail" on page 43 for details.

Excluding files and directories from the view

Exclusionary mappings allow files and directories to be excluded from a client workspace. This is accomplished by prefacing the mapping with a minus sign (-). Whitespace is not allowed between the minus sign and the mapping.

Example: *Using views to exclude files from a client workspace.*

Bill, whose client is named `billm`*, wants to view only source code; he's not interested in the documentation files. His client view would look like this:*

```
View:
        //depot/elm_proj/...          //billm/...
        -//depot/elm_proj/doc/...     //billm/doc/...
```

Since later mappings have precedence over earlier ones, no files from the depot's `doc` *subdirectory will ever be copied to Bill's client. Conversely, if Bill does have a* `doc` *subdirectory in his client, no files from that subdirectory will ever be copied to the depot.*

Allowing filenames in the client to differ from depot filenames

Mappings can be used to make the filenames in the client workspace differ from those in the depot.

Example: *Files with different names in the depot and client workspace.*

Mike wants to store the files as above, but he wants to take the `elmdoc.`*X files in the depot and call them* `helpfile.`*X in his client workspace. He uses the following mappings:*

```
View:
        //depot/elm_proj...                 //mike_elm/...
        //depot/elm_proj/doc/elmdoc.*       //mike_elm/help/helpfile.*
```

Each wildcard on the depot side of a mapping must have a corresponding wildcard on the client side of the same mapping. The wildcards are replaced in the copied-to direction by the substring that the wildcard represents in the copied-from direction.

There can be multiple wildcards; the *n*th wildcard in the depot specification corresponds to the *n*th wildcard in the client description.

Changing the order of filename substrings

The `%d` wildcard matches strings similarly to the `*` wildcard, but `%d` can be used to rearrange the order of the matched substrings.

Example: *Changing string order in client workspace names.*

Mike wants to change the names of any files with a dot in them within his `doc` *subdirectory in such a way that the file's suffixes and prefixes are reversed in his client workspace. For example, he'd like to rename the* `Elm.cover` *file in the depot* `cover.Elm` *in his client workspace. (Mike can be a bit difficult to work with). He uses the following mappings:*

```
View:
        //depot/elm_proj/...            //mike_elm/...
        //depot/elm_proj/doc/%1.%2      //mike_elm/doc/%2.%1
```

Two mappings can conflict and fail

When you use multiple mappings in a single view, some files may map to two separate places in the depot or on the client. When a file doesn't map to the same place in both directions, Perforce ignores that file.

Example: *Mappings that conflict and fail.*

Joe has constructed a view as follows:

```
View:
        //depot/proj1/...         //joe/proj1/...
        //depot/proj1/foo         //joe/proj1/bar
```

The depot file `//depot/proj1/bar` *maps to* `//joe/proj1/bar`*, but that same client file* `//joe/proj1/bar` *maps back to the depot via the higher-precedence second line to* `//depot/proj1/foo`*. Because the file would be written back to a different location in the depot than where it was read from, Perforce doesn't map this name at all, and the file is ignored.*

In older versions of Perforce, this was often used as a trick to exclude particular files from the client workspace. Because Perforce now has exclusionary mappings, this type of mapping is no longer useful, and should be avoided.

Editing Existing Client Specifications

You can use `p4 client` at any time to change the client workspace specification.

Note that changing a client specification has no immediate effect on the locations of any files; the locations of files in your workspace are affected only when the client specification is *used* in subsequent commands.

This is particularly important for two types of client spec changes, specifically changes to the client view and changes to the client root:

- If you change only the value of the client workspace `View:` field with `p4 client`, perform a `p4 sync` immediately afterward. The files in the client workspace will be deleted from their old locations and written to their new locations.
- If you use `p4 client` to change the client workspace `Root:`, use `p4 sync #none` to remove the files from their old location in the workspace *before* using `p4 client` to change the client root. After using `p4 client` to change the root directory, perform a `p4 sync` to copy the files to their new locations within the client view. (If you forget to perform the `p4 sync #none` before changing the client view, you can always remove the files from their client workspace locations manually).

Warning! It's not a good idea to change both the client `Root:` and the client `View:` at the same time. Change either the `Root:` or the `View:` according to the instructions above, make sure that the files are in place in the client workspace, and then change the other.

Deleting an existing client specification

An existing client workspace specification can be deleted with `p4 client -d` *`clientname`*. Deleting a client specification has no effect on any files in the client workspace or depot; it simply removes the Perforce server's record of the mapping between the depot and the client workspace.

To delete existing files from a client workspace, use `p4 sync #none` (described in "Specifying Older File Revisions" on page 51) on the files *before* deleting the client specification, or use the standard local OS deletion commands *after* deleting the client specification.

Client specification options

The `Options:` field contains six values, separated by spaces. Each of the six options have two possible settings.

The following table provides the option values and their meanings:

Option	Choice	Default
`[no]allwrite`	Should unopened files be left writable on the client?	`noallwrite`
`[no]clobber`	Should `p4 sync` overwrite (clobber) writable but unopened files in the client with the same name as the newly synced files?	`noclobber`
`[no]compress`	Should the data sent between the client and the server be compressed? Both client and server must be version 99.1 or higher, or this setting will be ignored.	`nocompress`
`[no]crlf`	***Note***: *2000.2 or earlier only!* Should CR/LF translation be performed automatically when copying files between the depot and the client workspace? (On UNIX, this setting is ignored).	`crlf`
`[un]locked`	Do other users have permission to edit the client specification? (To make a locked client specification truly effective, be sure to set a password for the client's owner with `p4 passwd`.) If `locked`, only the owner is able to use, edit, or delete the client spec. Note that a Perforce administrator is still able to override the lock with the `-f` (force) flag.	`unlocked`

Option	Choice	Default
`[no]modtime`	For files *without* the `+m` (modtime) file type modifier: • For Perforce clients at the 99.2 level or earlier, if `modtime` is set, the modification date (on the local filesystem) of a newly synced file is the date and time *at the server* when the file was submitted to the depot. • For Perforce clients at the 2000.1 level or higher, if `modtime` is set, the modification date (on the local filesystem) of a newly synced file is the datestamp *on the file* when the file was submitted to the depot. • If `nomodtime` is set, the modification date is the date and time of sync, regardless of Perforce client version. For files *with* the `+m` (modtime) file type modifier: • For Perforce clients at the 99.2 level or earlier, the `+m` modifier is ignored, and the behavior of `modtime` and `nomodtime` is as documented above. • For Perforce clients at the 2000.1 level or higher, the modification date (on the local filesystem) of a newly synced file is the datestamp *on the file* when the file was submitted to the depot, *regardless* of the setting of `modtime` or `nomodtime` on the client.	`nomodtime` (i.e. date and time of sync) for most files. Ignored for files with the `+m` file type modifier.
`[no]rmdir`	Should `p4 sync` delete empty directories in a client if all files in the directory have been removed?	`normdir`

Multiple workspace roots for cross-platform work

Users who work on more than one operating system, but who wish to use the same client workspace, can use the `AltRoots:` field in the `p4 client` form.

Up to two alternate client workspace roots may be specified. Perforce compares the current working directory against the main `Root:` first, and then against the two `AltRoots:` if they exist. The first root to match the current working directory is used. If no roots match, the main root is used.

If `edk`'s current working directory is under `e:\porting\edk\elm`, then Perforce will use the Windows path as his client workspace root, rather than `/usr/edk/elm`. This allows

`edk` to use the same client workspace specification for both UNIX and Windows development.

```
Client: eds_elm
Owner:  edk
Description:
        Created by ed.
Root:   /usr/edk/elm
AltRoots:
        e:\porting\edk\elm
Options:        nomodtime noclobber
View:
        //depot/...    //eds_elm/...
```

If you are using multiple client workspace roots, you can always find out which workspace root is in effect by examining the `Client root:` as reported by `p4 info`.

Line-ending conventions (CR/LF translation)

The `LineEnd:` field controls the line-ending character(s) used for text files in the client workspace.

Note The `LineEnd:` option is new to Perforce 2001.1, and replaces the old convention of specifying `crlf` or `nocrlf` in the `Options:` field.

The `LineEnd:` field accepts one of five values:

Option	Meaning
`local`	Use mode native to the client (default)
`unix`	UNIX-style line endings: `LF` only
`mac`	Macintosh-style: `CR` only
`win`	Windows-style: `CR`, `LF`
`share`	Shared mode: Line endings are `LF` with any `CR/LF` pairs translated to `LF`-only style before storage or syncing with the depot. When you sync your client workspace, line endings will be `LF`. If you edit the file on a Windows machine, and your editor inserts `CR`s before each `LF`, the extra `CR`s will not appear in the archive file. The most common use of the `share` option is for users of Windows workstations who mount their UNIX home directories mounted as network drives; if you sync files from UNIX, but edit the files on a Windows machine, the `share` option eliminates problems caused by Windows-based editors' insertion of carriage returns in text files.

Referring to Files on the Command Line

File names provided as arguments to Perforce commands can be referred to in one of two ways: by using the names of the files in the client workspace, or by providing the names of the files in the depot. When providing client workspace file names, you may give the name in either *local* syntax or Perforce syntax.

Local syntax

Local syntax is simply a file's name as specified by the local shell or OS. This name may be an absolute path, or may be specified relative to the current directory, although it can only contain relative components at the beginning of the file name (that is, it doesn't allow `sub/dir/./here/foo.c`). On UNIX, Ed could refer to the `README` file at Elm's top level as `/usr/edk/elm/README`, or in a number of other ways.

Perforce syntax

Perforce provides its own filename syntax which remains the same across operating systems. Filenames specified in this way begin with two slashes and the client or depot name, followed by the path name of the file relative to the client or depot root directory.

The components of the path are separated by slashes.

Examples of Perforce syntax

```
//depot/...
//elm_client/docs/help.1
```

Perforce syntax is sometimes called *depot syntax* or *client syntax*, depending on whether the file specifier refers to a file in the depot or on the client, but both forms of file specification work the same way.

Providing files as arguments to commands

Because the client view provides a one-to-one mapping between any file in the client workspace and any file in the depot, *any file can be specified within any PERFORCE command in client syntax, depot syntax, or local syntax*. A depot's file specifier can be used to refer to a file in the client, and vice versa. Perforce will do the necessary mapping to determine which file is actually used.

Any filenames provided to Perforce commands can be specified in any valid local syntax, or in Perforce syntax by depot or client. If a client filename is provided, Perforce uses the client view to locate the corresponding file in the depot. If a depot filename is given, the client view is used to locate the corresponding file in the client workspace.

Example: *Uses of different syntaxes to refer to a file.*

Ed wants to delete the `src/lock.c` *file. He can give the* `p4 delete` *command in a number of ways:*

While in his client root directory, he could type

```
p4 delete src/lock.c
```

or, while in the `src` *subdirectory, he could type*

```
p4 delete lock.c
```

or, while in any directory on the client host, he could type any of the following commands:

```
p4 delete //eds_elm/src/lock.c
p4 delete //depot/elm_proj/src/lock.c
p4 delete /usr/edk/elm/src/lock.c
```

Client names and depot names in a single Perforce server share the same namespace, so Perforce will never confuse a client name with a depot name. Client workspace names and depot names can never be the same.

Note The point of this section is worth repeating: any file can be specified within any Perforce command in client syntax, depot syntax, or local syntax. The examples in this manual will use these syntaxes interchangeably.

Wildcards and Perforce syntax

Perforce wildcards may be mixed with both local or Perforce syntax. For example:

Wildcard	Meaning
`J*`	Files in the current directory starting with `J`
`*/help`	All files called help in current subdirectories
`...`	All files under the current directory and its subdirectories
`.../*.c`	All such files ending in `.c`
`/usr/edk/...`	All files under `/usr/edk`
`//weasel/...`	All files on client (or depot) `weasel`
`//depot/...`	All files in the depot named `depot`
`//...`	All files in all depots (when used to specify files on the command line)

Name and String Limitations

Illegal characters in filenames and Perforce objects

In order to support internationalization, Perforce allows the use of "unprintable" (i.e. non-ASCII) characters in filenames, label names, client workspace names, and other identifiers.

Perforce's wildcards and the revision-specifying characters `@` and `#`, are not allowed in file names, label names, or other identifiers:

Character	Reason
`@`	Perforce revision specifier for date, label name, or changelist number.
`#`	Perforce revision specifier for revision numbers.
`*`	Perforce wildcard: matches anything, works within a single directory
`...`	Perforce wildcard: matches anything, works at the current directory level and includes files in all directory levels below the current level.
`%`	Perforce wildcard: `%0` through `%9` are used for positional substitutions.
`/`	Perforce separator for pathname components.

Observe that most of these characters tend to be difficult to use in filenames in cross-platform environments: UNIX separates path components with `/`, while many DOS commands interpret `/` as a command line switch. Most UNIX shells interpret `#` as the beginning of a comment. Both DOS and UNIX shells automatically expand `*` to match multiple files, and the DOS command line uses `%` to refer to variables.

Caveats on non-ASCII filenames

Although non-ASCII characters are allowed in filenames and Perforce identifiers, entering them from the command line may require platform-specific solutions. Users of GUI-based file managers can manipulate such files with drag-and-drop operations.

For internationalization purposes, there are some limitations on how filenames with non-ASCII character sets are displayed. For Release 2001.1, all users should use a common code page setting (under Windows, use the **Regional Settings** applet in the **Control Panel**; under UNIX, set the `LOCALE` environment variable) in order to ensure that all filenames are displayed consistently across all machines in your organization.

If you are using Perforce in internationalized mode, all users must have `P4CHARSET` set properly. For details, see the *Command Reference*.

Caveats on using spaces in filenames

Filenames can include spaces, but require some care as to their specification in client views and root mappings. If you use spaces in a client view or client root, quote the string in which the space appears.

Make sure not to quote spaces that divide two separate strings; for example, a client mapping containing a directory named "`my projects`" would be quoted like this:

```
//depot/projects/foo/... "//aclient/my projects/foo/..."
```

Other Perforce objects, such as branch names, client names, label names, and so on, may be *specified* with spaces, but these spaces will be automatically converted to underscores by the Perforce server.

Name and description lengths

Descriptions in the forms used by `p4 client`, `p4 branch`, and so on, may be of any length. Similarly, all names given to Perforce objects such as branches, clients, and so on, are also limited to 1024 characters.

Specifying Older File Revisions

All of the commands and examples we've seen thus far have been used to operate only on the most recent revisions of particular files, but many Perforce commands can act on older file versions. For example, if Ed types `p4 sync //eds_elm/src/lock.c`, the latest revision, or *head revision*, of `lock.c` is retrieved, but older revisions can be retrieved by tacking a revision specification onto the end of the file name.

Warning! Some OS shells treat the revision character # as a comment character if it starts a new word. If your shell is one of these, escape the # before use.

Revision Specifier	Meaning	Examples
file`#`*n*	Revision number	`p4 sync lock.c#3` Refers to revision 3 of file `lock.c`

file`@`*n*	A change number	`p4 sync lock.c@126`
		Refers to the version of `lock.c` when changelist 126 was submitted, even if it was not part of the change.
		`p4 sync //depot/...@126`
		Refers to the state of the entire depot at changelist 126 (numbered changelists are explained in Chapter 7, Changelists).

file`@`*labelname*	A label name	`p4 sync lock.c@beta` The revision of `lock.c` in the label called `beta` (labels are explained in Chapter 8, Labels).

Revision Specifier	Meaning	Examples
file`@`*clientname*	A client name. The revision of *file* last taken into client workspace *clientname*.	`p4 sync lock.c@lisag_ws` The revision of `lock.c` last taken into client workspace `lisag_ws`
file`#none`	The nonexistent revision.	`p4 sync lock.c#none` Says that there should be no version of `lock.c` in the client workspace, even if one exists in the depot.
file`#head`	The head revision, or latest version, of the file.	`p4 sync lock.c#head` Except for explicitly noted exceptions, this is identical to referring to the file with no revision specifier.
file`#have`	The revision on the current client. This is synonymous to `@client` where client is the current client name.	`p4 sync lock.c#have` The revision of `lock.c` found in the current client.
file`@`*date*	The head revision of the file at 00:00:00 on the morning of that date. Dates are specified as YYYY/MM/DD.	`p4 sync lock.c@1998/05/18` The head revision of `lock.c` as of 00:00:00, May 18, 1998.

Revision Specifier	Meaning	Examples

file@"*date time*"	The head revision of the file in the depot on the given date at the given time. The date is specified as above; the time is specified as HH:MM:SS.	`p4 sync lock.c@"1998/05/18 15:21:34"`
		`p4 sync lock.c@1998/05/18:15:21:34`
		The head revision of `lock.c` as of May 18, 1998, at 3:21:34 pm. Both forms shown above are equivalent.
		The date and the time must be separated by a single space or a colon, and the entire string should be quoted. The time is specified on the 24-hour clock.
		Warning! Perforce allows you to search on dates with two-digit years, but these years are assumed to fall in the twentieth century. For safety's sake, use four-digit years.

In all cases, if a file doesn't exist at the given revision number, it appears as if the file doesn't exist at all. Thus, using a label to refer to a file that isn't in the label is indistinguishable from referring to a file that doesn't exist at all.

Date and time specifications are always interpreted with respect to the local time zone of the Perforce server. Note that because the server stores times internally in terms of number of seconds since the Epoch (00:00:00 GMT Jan. 1, 1970), if you move your server across time zones, the times recorded on the server will automatically be reported in the new timezone.

The date, time, offset from GMT, and time zone in effect at your Perforce server are displayed in the "`Server date:`" line in the output of `p4 info`.

Using revision specifications without filenames

Revision specifications can be provided without file names. This limits the command's action to the specified revision of all files in the depot or in the client's workspace. Thus, `#head` refers to the head revisions of all files in the depot, and `@`*`labelname`* refers to the revisions of all files in the named label.

Example: *Retrieving files using revision specifiers.*

Ed wants to retrieve all the doc files into his Elm `doc` *subdirectory, but he wants to see only those revisions that existed at change number 30. He types*

```
p4 sync //eds_elm/doc/*@30
```

Later, he creates another client for a different user. The new client should have all of the file revisions that Ed last synced. Ed sets up the new client specification and types

```
p4 sync //depot/elm_proj/...@eds_elm
```

He could have typed

```
p4 sync @eds_elm
```

and the effect would have been the same.

Example: *Removing files from the client workspace.*

Another client needs all its files removed, but these files shouldn't be deleted from the depot. Ed sets `P4CLIENT` *to the correct clientname, and types*

```
p4 sync #none
```

The files are removed from his workspace ("synced to the nonexistent revision"), but not from the depot.

Revision Ranges

A few Perforce client commands can limit their actions to a range of revision numbers, rather than just a single revision. A revision range is two revision specifications, separated by a comma.

If only a single revision is given where a revision range is expected, the named revision specifies the end of the range, and the start of the range is assumed to be 1. If no revision number or range is given where a revision range is expected, the default is all revisions.

The commands that accept revision range specifications are:

`p4 changes`	`p4 file`	`p4 integrate`	`p4 jobs`
`p4 print`	`p4 sync`	`p4 verify`	

Example: *Listing changes with revision ranges.*

A release manager needs to see a list of all changes to the Elm project in July. He types:

```
p4 changes //depot/elm_proj/...@2000/7/1,2000/8/1
```

The resulting list of changes looks like this:

```
Change 632 on 2000/07/1 by edk@eds_elm 'Started work'
Change 633 on 2000/07/1 by edk@eds_elm 'First build w/bug fix'
...
Change 673 on 2000/07/31 by edk@eds_elm 'Final build for QA'
```

He can then use `p4 describe` *change#* *against any desired change for a full description.*

File Types

Perforce supports six base file types: `text` files, `binary` files, `unicode` files, native `apple` files on the Macintosh, Mac `resource` forks, and UNIX `symlinks`. File type modifiers are then applied to the base types allowing for support of RCS keyword expansion, file compression on the server, and more.

Perforce attempts to determine the type of the file automatically: when a file is opened with `p4 add`, Perforce first decides if the file is a regular file or a symbolic link, and then examines the first part of the file to determine whether it's `text` or `binary`. If any non-text characters are found, the file is assumed to be `binary`; otherwise, the file is assumed to be `text`. (Files of type `unicode` are detected only when the server is running in internationalized mode; for details, see your system administrator.)

Some file formats (for example, Adobe PDF files, and Rich Text Format files) are actually `binary` files, but can sometimes be mistakenly detected by Perforce as being of type `text`. For these files, your system administrator can use the `p4 typemap` command to set up a table matching Perforce file types to file name specifications.

Whenever you open a new file for `add`, Perforce checks the typemap table. If the file matches an entry in the table, Perforce uses the file type specified in the table, rather than the file type it would have otherwise used. You can override the file type specified in the typemap table by specifying it on the command line with the `-t` *`filetype`* modifier.

Once set, a file's type is normally inherited from one revision to the next, but can be overridden or changed with the `-t` flag:

- `p4 add -t` *`filetype filespec`* adds the files as the type you've specified.
- `p4 edit -t` *`filetype filespec`* opens the file for `edit`; when the files are submitted, their type will be changed.
- `p4 reopen -t` *`filetype filespec`* changes the type of a file that's already open for `add` or `edit`.

The *`filetype`* argument is specified as *`basetype+modifiers`*. File type modifiers may be combined; for instance, to change the file type of your Perl script `myscript.pl` to executable text with RCS keyword expansion, use `p4 edit -t text+kx myscript.pl`. Full listings of the base file types and their modifiers are provided below.

The type of an existing file can be determined with `p4 opened` or `p4 files`.

File revisions of binary files are normally stored in full within the depot, but only changes made to text files since the previous revision are normally stored. This is called *delta storage*, and Perforce uses RCS format to store its deltas. The file's type determines whether *full file* or *delta* storage is used. When delta storage is used, file merges and file compares can be performed. Files that are stored in their full form can't be merged or

compared. (RCS format and delta storage are described in more detail at the start of the next chapter).

Some of the file types are compressed to `gzip` format for storage in the depot. The compression occurs during the submission process, and decompression happens while syncing. The client workspace always contains the file as it was submitted.

Warning! Do not try to fool Perforce into storing binary files in delta storage by changing the file type to `text`! If you add a file that contains a `^Z` as text from a Windows client, only the part of the file up to the `^Z` will be stored in the depot.

Note Versions of Perforce prior to 99.1 used a very different, somewhat limited, set of file types. These types have been maintained for backwards compatibility.

Mappings between the old and new file types are described in the *Perforce Command Reference*.

The base Perforce file types are:

Keyword	Description	Comments	Server Storage Type
`text`	Text file	Treated as text on the client. Line-ending translations are performed automatically on Windows and Macintosh clients.	delta
`binary`	Non-text file	Accessed as binary files on the client. Stored compressed within the depot.	full file, compressed
`symlink`	Symbolic link	UNIX clients (and the BeOS client) access these as symbolic links. Non-UNIX clients treat them as (small) text files.	delta
`apple`	Multi-forked Macintosh file	AppleSingle storage of Mac data fork, resource fork, file type and file creator. New to Perforce 99.2. For full details, please see the Mac platform notes at `http://www.perforce.com/perforce/technical.html`	full file, compressed, AppleSingle format.

Keyword	Description	Comments	Server Storage Type
`resource`	Macintosh resource fork	The only file type for Mac resource forks in Perforce 99.1 and before. This is still supported, but we recommend using the new `apple` file type instead. For full details, please see the Mac platform notes at `http://www.perforce.com/perforce/technical.html`	full file, compressed
`unicode`	Unicode file	Perforce servers operating in internationalized mode support a Unicode file type. These files are translated into the local character set. For details, see the *System Administrator's Guide*.	Stored as UTF-8

The file type modifiers are:

Modifier	Description	Comments
`+x`	Execute bit set on client	Used for executable files.
`+w`	File is always writable on client	
`+ko`	Old-style keyword expansion	Expands only the `$Id$` and `$Header$` keywords: This pair of modifiers exists primarily for backwards compatibility with versions of Perforce prior to 2000.1, and corresponds to the `+k` (`ktext`) modifier in earlier versions of Perforce.

Modifier	Description	Comments
+k	RCS keyword expansion	Expands RCS (Revision Control System) keywords. RCS keywords are case-sensitive. When using keywords in files, a colon after the keyword (e.g., $Id:$) is optional. Supported keywords are: • Id • $Header$ • $Date$ • $DateTime$ • $Change$ • $File$ • $Revision$ • $Author$
+l	Exclusive open (locking)	If set, only one user at a time will be able to open a file for editing. Useful for binary file types (e.g., graphics) where merging of changes from multiple authors is meaningless.
+C	Server stores the full compressed version of each file revision	Default server storage mechanism for binary files.
+D	Server stores deltas in RCS format	Default server storage mechanism for text files.
+F	Server stores full file per revision	Useful for long ASCII files that aren't read by users as text, such as PostScript files.
+S	Only the head revision is stored on the server	Older revisions are purged from the depot upon submission of new revisions. Useful for executable or .obj files.
+m	Preserve original modtime	The file's timestamp on the local filesystem is preserved upon submission and restored upon sync. Useful for third-party DLLs in Windows environments.

The modtime (+m) modifier is a special case: It is intended for use by developers who need to preserve a file's original timestamp. (Normally, Perforce updates the timestamp when a file is synced.) It allows a user to ensure that the timestamp of a file in a client workspace after a p4 sync will be the original timestamp existing *on the file* at the time of submission

(that is, *not* the time at the Perforce server at time of submission, and *not* the time on the client at the time of sync).

The most common case where this is useful is development involving the third-party DLLs often encountered in Windows environments. Because the timestamps on such files are often used as proxies for versioning information (both within the development environment and also by the operating system), it is sometimes necessary to preserve the files' original timestamps regardless of a Perforce user's client settings.

The `+m` modifier on a file allows this to happen; if set, Perforce ignores the `modtime` ("file's timestamp at time of submission") or `nomodtime` ("date and time on the client at time of sync") setting at the `p4 client` level when syncing the file, and always restore the file's original timestamp at the time of submit.

RCS keywords are expanded as follows:

Keyword	Expands To	Example
`$Id$`	File name and revision number in depot syntax	`$Id: //depot/path/file.txt#3 $`
`$Header$`	Synonymous with `$Id$`	`$Header: //depot/path/file.txt#3 $`
`$Date$`	Date of last submission in format *YYYY/MM/DD*	`$Date: 2000/08/18 $`
`$DateTime$`	Date and time of last submission in format *YYYY/MM/DD hh:mm:ss* Date and time are as of the local time on the Perforce server at time of submission.	`$DateTime: 2000/08/18 23:17:02 $`
`$Change$`	Perforce changelist number under which file was submitted	`$Change: 439 $`
`$File$`	File name only, in depot syntax (without revision number)	`$File: //depot/path/file.txt $`
`$Revision$`	Perforce revision number	`$Revision: #3 $`
`$Author$`	Perforce user submitting the file	`$Author: edk $`

Forms and Perforce Commands

Certain Perforce commands, such as `p4 client` and `p4 submit`, present a form to the user to be filled in with values. This form is displayed in the editor defined in the environment variable `P4EDITOR`. When the user changes the form and exits the editor, the form is parsed by Perforce, checked for errors, and used to complete the command operation. If there are errors, Perforce gives an error message and you must try again.

The rules of form syntax are simple: keywords must be against the left margin and end with a colon, and values must either be on the same line as the keyword or indented on the lines beneath the keyword. Only the keywords already present on the form are recognized. Some keywords, such as the `Client:` field in the `p4 client` form, take a single value; other fields, such as `Description:`, take a block of text; and others, like `View:`, take a list of lines.

Certain fields, like `Client:` in `p4 client`, can't have their values changed; others, like `Description:` in `p4 submit`, must have their values changed. If you don't change a field that needs to be changed, or vice versa, the worst that will happen is that you'll get an error. We've done our best to make these cases as self-evident as possible. When in doubt, consult the *Perforce Command Reference* or use `p4 help` *`command`*.

Reading forms from standard input; Writing forms to standard output

Any commands that require the user to fill in a form, such as `p4 client` and `p4 submit`, can read the form from standard input with the `-i` flag. Similarly, the `-o` flag can be used to write a form specification to standard output.

These two flags are primarily used in scripts that access Perforce: use the `-o` flag to read a form, process the strings representing the form within your script, and finally, use the `-i` flag to send the processed form back to Perforce.

For example, to create a new job within a script you could first use `p4 job -o >` *`tempfile`* to read a blank job specification, then add information to the proper lines in *`tempfile`*, and finally use a command like `p4 job -i <` *`tempfile`* to store the new job in Perforce.

The commands that display forms and can therefore use these flags are:

`p4 branch`	`p4 change`	`p4 client`
`p4 job`	`p4 label`	`p4 protect`
`p4 submit`*	`p4 typemap`	`p4 user`

* `p4 submit` can take the `-i` flag, but not the `-o` flag.

General Reporting Commands

Many reporting commands have specialized functions, and these are discussed in later chapters. The following reporting commands give the most generally useful information; all of these commands can take file name arguments, with or without wildcards, to limit reporting to specific files. Without the file arguments, the reports are generated for all files.

These reports always generate information on depot files, not files within the client workspace. As with any other Perforce command, when a client file is provided on the command line, Perforce maps it to the proper depot file.

Command	Meaning
`p4 filelog`	Generates a report on each revision of the file(s), in reverse chronological order.
`p4 files`	Lists file name, latest revision number, file type, and other information about the named file(s).
`p4 sync -n`	Tells you what `p4 sync` would do, without doing it.
`p4 have`	Lists all the revisions of the named files within the client that were last gotten from the depot. Without any files specifier, it lists all the files in the depot that the client has.
`p4 opened`	Reports on all files in the depot that are currently open for edit, add, delete, branch, or integrate within the client workspace.
`p4 print`	Lists the contents of the named file(s) to standard output.
`p4 where`	Given a file argument, displays the mapping of that file within the depot, the client workspace, and the local OS.

Revision specifiers can be used with all of these reporting commands, for example `p4 files @clientname` can be used to report on all the files in the depot that are currently found in client *clientname*. See Chapter 11, Reporting and Data Mining, for a more detailed discussion of each of these commands.

Chapter 5 Perforce Basics: Resolving File Conflicts

File conflicts can occur when two users edit and submit two versions of the same file. Conflicts can occur in a number of ways, but the situation is usually a variant of the following:

1. Ed opens file `foo` for edit.
2. Lisa opens the same file in her client for edit.
3. Ed and Lisa both edit their client workspace versions of `foo`.
4. Ed submits a changelist containing `foo`, and the submit succeeds.
5. Lisa submits a changelist with her version of `foo`; and her submit fails.

If Perforce were to accept Lisa's version into the depot, the head revision would contain none of Ed's changes. Instead, the changelist is rejected and a resolve must be performed. The resolve process allows a choice to be made: Lisa's version can be submitted in place of Ed's, Lisa's version can be dumped in favor of Ed's, a Perforce-generated merged version of both revisions can be submitted, or the Perforce-generated merged file can be edited and then submitted.

Resolving a file conflict is a two-step process: first the resolve is *scheduled*, then the resolve is *performed*. A resolve is automatically scheduled when a submit of a changelist fails because of a file conflict; the same resolve can be scheduled manually, without submitting, by syncing the head revision of a file over an opened revision within the client workspace. Resolves are always performed with `p4 resolve`.

Perforce also provides facilities for locking files when they are edited. This can eliminate file conflicts entirely.

RCS Format: How Perforce Stores File Revisions

Perforce uses RCS format to store its text file revisions; binary file revisions are always saved in full. If you already understand what this means, you can skip to the next section of this chapter, as the remainder of this section explains how RCS format works.

Only the differences between revisions are stored

A single file might have hundreds, even thousands, of revisions. Every revision of a particular file must be retrievable, and if each revision was stored in full, disk space problems could occur: one thousand 10KB files, each with a hundred revisions, would use a gigabyte of disk space. The scheme used by most SCM systems, including Perforce, is to

save only the latest revision of each file, and then store the differences between each file revision and the one previous.

As an example, suppose that a Perforce depot has three revisions of file `foo`. The head revision (`foo#3`) looks like this:

```
foo#3:
This is a test
of the
emergency
broadcast system
```

Revision two might be stored as a symbolic version of the following:

```
foo#2:
line 3 was "urgent"
```

And revision one might be a representation of this:

```
foo#1:
line 4 was "system"
```

From these partial file descriptions, any file revision can be reconstructed. The reconstructed `foo#1` would read

```
Reconstructed foo#1:
This is a test
of the
urgent
system
```

The RCS (*Revision Control System*) algorithm, developed by Walter Tichy, uses a notation for implementing this system that requires very little storage space and is quite fast. In RCS terminology, it is said that the full text of the head revisions are stored, along with the reverse deltas of each previous revision.

It is interesting to note that the full text of the first revision could be stored, with the deltas leading forward through the revision history of the file, but RCS has chosen the other path: the full text of the head revision of each file is stored, with the deltas leading backwards to the first revision. This is because the head revision is accessed much more frequently than previous file revisions; if the head revision of a file had to be calculated from the deltas each time it was accessed, any SCM utilizing RCS format would run much more slowly.

Use of “diff” to determine file revision differences

RCS uses the “GNU diff” algorithm to determine the differences between two versions of the same file; p4d contains its own diff routine which is used by Perforce servers to determine file differences when storing deltas.

Because Perforce’s diff always determines file deltas by comparing chunks of text between newline characters, it is by default only used with `text` files. If a file is `binary`, each revision is stored in full.

Scheduling Resolves of Conflicting Files

Whenever a file revision is to be submitted that is not an edit of the file’s current head revision, there will be a file conflict, and this conflict must be resolved.

In slightly more technical terms, we’ll call the file revision that was read into a client workspace the *base file revision*. If the base file revision for a particular file in a client workspace is not the same as the head revision of the same file in the depot, a *resolve* must be performed before the new file revision can be accepted into the depot.

Before resolves can be performed with `p4 resolve`, they must be scheduled; this can be done with `p4 sync`. An alternative is to submit a changelist that contains the newly conflicting files. If a resolve is necessary, the submit fails, and the resolve is scheduled automatically.

Why “p4 sync” to Schedule a Resolve?

Remember that the job of `p4 sync` is to project the state of the depot onto the client. Thus, when `p4 sync` is performed on a particular file:

- If the file does not exist in the client, or it is found in the client but is unopened, it is copied from the depot to the client.
- If the file has been deleted from the depot, it is deleted from the client.
- If the file has been opened in the client with `p4 edit`, the Perforce server can’t simply copy the file onto the client: any changes that had been made to the current revision of the file in the client would be overwritten. Instead, a *resolve* is scheduled between the file revision in the depot, the file on the client, and the base file revision (the revision that was last read into the client).

Example: *Scheduling resolves with* `p4 sync`

Ed is making a series of changes to the `*.guide` *files in the elm doc subdirectory. He has retrieved the* `//depot/elm_proj/doc/*.guide` *files into his client and has opened the files with* `p4 edit`. *He edits the files, but before he has a chance to submit them, Lisa submits new*

versions of some of the same files to the depot. The versions Ed has been editing are no longer the head revisions, and resolves must be scheduled and performed for each of the conflicting files before Ed's edits can be accepted.

Ed schedules the resolves with `p4 sync //edk/doc/*.guide`*. Since these files are already open in the client, Perforce doesn't replace the client files. Instead, Perforce schedules resolves between the client files and the head revisions in the depot.*

Alternatively, Ed could have submitted the `//depot/elm_proj/doc/*.guide` *files in a changelist; the file conflicts would cause the* `p4 submit` *to fail, and the resolves would be scheduled as part of the submission failure.*

How Do I Know When a Resolve is Needed?

`p4 submit` fails when it determines that any of the files in the submitted changelist need to be resolved, and the error message includes the names of the files that need resolution. If the changelist provided to `p4 submit` was the default changelist, it is assigned a number, and this number must be used in all future references to the changelist. (Numbered changelists are discussed in Chapter 7, Changelists)

Another way of determining whether a resolve is needed is to run `p4 sync -n` *`filenames`* before performing the submit, using the files in the changelist as arguments to the command. If file conflict resolutions are necessary, `p4 sync -n` reports them. The advantage of this approach is that the files in the default changelist remain in the default changelist (that is, the default changelist will not be reassigned to a numbered changelist).

Performing Resolves of Conflicting Files

File conflicts are fixed with `p4 resolve [`*`filenames`*`]`. Each file provided as an argument to `p4 resolve` is processed separately. `p4 resolve` starts with three revisions of the same file and generates a fourth version; the user can accept any of these revisions in place of the current client file, and can edit the generated version before accepting it. The new revisions must then be submitted with `p4 submit`.

`p4 resolve` is interactive; a series of options are displayed for the user to respond to. The dialog looks something like this:

```
/usr/edk/elm/doc/answer.1 - merging //depot/elm_proj/doc/answer.1#5
Diff chunks: 4 yours + 2 theirs + 1 both + 1 conflicting
Accept(a) Edit(e) Diff(d) Merge (m) Skip(s) Help(?) [e]:
```

The remainder of this section explains what this means, and how to use this dialog.

File revisions used and generated by "p4 resolve"

`p4 resolve [filenames]` starts with three revisions of the same file, generates a new version that merges elements of all three revisions, allows the user to edit the new file, and writes the new file (or any of the original three revisions) to the client. The file revisions used by `p4 resolve` are these:

yours	The newly-edited revision of the file in the client workspace. This file is overwritten by *result* once the resolve process is complete.
theirs	The revision in the depot that the client revision conflicts with. Usually, this is the head revision, but `p4 sync` can be used to schedule a resolve with any revision between the head revision and *base*.
base	The file revision in the depot that *yours* was edited from. Note that *base* and *theirs* are different revisions; if they were the same, there would be no reason to perform a resolve.
merge	File variation generated by Perforce from *theirs*, *yours*, and *base*.
result	The file resulting from the resolve process. *result* is written to the client workspace, overwriting *yours*, and must subsequently be submitted by the user. The instructions given by the user during the resolve process determine exactly what is contained in this file. The user can simply accept *theirs*, *yours*, or *merge* as the result, or can edit *merge* to have more control over the result.

The remainder of this chapter will use the terms *theirs*, *yours*, *base*, *merge*, and *result* to refer to the corresponding file revisions. The definitions given above are somewhat different when resolve is used to integrate *branched* files.

Types of conflicts between file revisions

The diff program that underlies the Perforce resolve mechanism determines differences between file revisions on a line-by-line basis. Once these differences are found, they are grouped into *chunks*: for example, three new lines that are adjacent to each other are grouped into a single chunk. *Yours* and *theirs* are both generated by a series of edits to *base*; for each set of lines in *yours*, *theirs*, and *base*, `p4 resolve` asks the following questions:

- Is this line set the same in *yours*, *theirs*, and *base*?
- Is this line set the same in *theirs* and *base*, but different in *yours*?

- Is this line set the same in *yours* and *base*, but different in *theirs*?
- Is this line set the same in *yours* and *theirs*, but different in *base*?
- Is this line set different in all three files?

Any line sets that are the same in all three files don't need to be resolved. The number of line sets that answer the other four questions are reported by `p4 resolve` in this form:

```
2 yours + 3 theirs + 1 both + 5 conflicting
```

In this case, two line sets are identical in *theirs* and *base* but are different in *yours*; three line sets are identical in *yours* and *base* but are different in *theirs*; one line set was changed identically in *yours* and *theirs*; and five line sets are different in *yours*, *theirs*, and *base*.

How the merge file is generated

`p4 resolve` generates a preliminary version of the *merge* file, which can be accepted as is, edited and then accepted, or rejected. A simple algorithm is followed to generate this file: any changes found in *yours*, *theirs*, or both *yours* and *theirs* are applied to the base file and written to the merge file; and any conflicting changes will appear in the merge file in the following format:

```
>>>> ORIGINAL VERSION foo#n
(text from the original version)
==== THEIR VERSION foo#m
(text from their file)
==== YOUR VERSION foo
(text from your file)
<<<<
```

Thus, editing the Perforce-generated merge file is often as simple as opening the merge file, searching for the difference marker "`>>>>`", and editing that portion of the text. This isn't always the case, as it is often necessary to examine the changes made to *theirs* to make sure they're compatible with other changes that you made. This can be facilitated by calling `p4 resolve` with the `-v` flag; `p4 resolve -v` tells Perforce to generate difference markers for all changes made in either file being resolved, instead of only for changes that are in conflict between the *yours* and *theirs* files.

The "p4 resolve" options

The `p4 resolve` command offers the following options:

Option	Short Meaning	What it Does
`e`	edit merged	Edit the preliminary merge file generated by Perforce
`ey`	edit *yours*	Edit the revision of the file currently in the client

Option	Short Meaning	What it Does
`et`	edit *`theirs`*	Edit the revision in the depot that the client revision conflicts with (usually the head revision). This edit is read-only.
`dy`	diff *`yours`*	Diff line sets from *`yours`* that conflict with *`base`*
`dt`	diff *`theirs`*	Diff line sets from *`theirs`* that conflict with *`base`*
`dm`	diff *`merge`*	Diff line sets from *`merge`* that conflict with *`base`*
`d`	diff	Diff line sets from *`merge`* that conflict with *`yours`*
`m`	merge	Invoke the command `P4MERGE` *`base theirs yours merge`*. To use this option, you must set the environment variable `P4MERGE` to the name of a third-party program that merges the first three files and writes the fourth as a result.
`?`	help	Display help for `p4 resolve`
`s`	skip	Don't perform the resolve right now.
`ay`	accept *`yours`*	Accept *`yours`* into the client workspace as the resolved revision, ignoring changes that may have been made in *`theirs`*.
`at`	accept *`theirs`*	Accept *`theirs`* into the client workspace as the resolved revision. The revision that was in the client workspace is overwritten.
`am`	accept *`merge`*	Accept *`merge`* into the client workspace as the resolved revision. The version originally in the client workspace is overwritten.
`ae`	accept edit	If you edited the *`merge`* file (by selecting "`e`" from the `p4 resolve` dialog), accept the edited version into the client workspace. The version originally in the client workspace is overwritten.
`a`	accept	If *`theirs`* is identical to *`base`*, accept *`yours`*, if *`yours`* is identical to *`base`*, accept *`theirs`*, if *`yours`* and *`theirs`* are different from *`base`*, and there are no conflicts between *`yours`* and *`theirs`*; accept merge, otherwise, there are conflicts between *`yours`* and *`theirs`*, so skip this file

Only a few of these options are visible on the command line, but all options are always accessible and can be viewed by choosing `help`. The *`merge`* file is generated by `p4d`'s internal diff routine. But the differences displayed by `dy`, `dt`, `dm`, and `d` are generated by a diff internal to the Perforce client program, and this diff can be overridden by specifying an external diff in the `P4DIFF` environment variable.

The command line has the following format:

```
Accept(a) Edit(e) Diff(d) Merge (m) Skip(s) Help(?) [am]:
```

Perforce's recommended choice is displayed in brackets at the end of the command line. Pressing `Return` or choosing `Accept` will perform the recommended choice. The recommended command is chosen by Perforce by the following algorithm: if there were no changes to *`yours`*, accept *`theirs`*. If there were no changes to *`theirs`*, accept *`yours`*. Otherwise, accept `merge`.

Example: *Resolving File Conflicts*

In the last example, Ed scheduled the `doc/*.guide` *files for resolve. This was necessary because both he and Lisa had been editing the same files; Lisa had already submitted versions, and Ed needs to reconcile his changes with Lisa's. To perform the resolves, he types* `p4 resolve //depot/elm_proj/doc/*.guide`*, and sees the following:*

```
/usr/edk/elm/doc/Alias.guide - merging
//depot/elm_proj/doc/Alias.guide#5
Diff chunks: 4 yours + 2 theirs + 1 both + 1 conflicting
Accept(a) Edit(e) Diff(d) Merge (m) Skip(s) Help(?) [e]:
```

This is the resolve dialog for `doc/Alias.guide`*, the first of the four* `doc/*.guide` *files. Ed sees that he's made four changes to the base file that don't conflict with any of Lisa's changes. He also notes that Lisa has made two changes that he's unaware of. He types* `dt` *(for "*`display` *`theirs`*`"`*) to view Lisa's changes; he looks them over and sees that they're fine. Of most concern to him, of course, is the one conflicting change. He types* `e` *to edit the Perforce-generated merge file and searches for the difference marker "*`>>>>`*". The following text is displayed:*

```
Acme Technology
Mountain View, California
>>>> ORIGINAL VERSION
==== THEIR VERSION
94041
==== YOUR VERSION
98041
<<<<
```

He and Lisa have both tried to add a zip code to an address in the file, but Ed had typed it wrong. He edits this portion of the merge file so it reads as follows:

```
Acme Technology
Mountain View, California
94041
```

The merge file is now acceptable to him: he's viewed Lisa's changes, seen that they're compatible with his own, and the only line conflict has been resolved. He quits from the editor and types `am`*; the edited merge file is written to the client, and the resolve process continues on the next* `doc/*.guide` *file.*

When a version of the file is accepted onto the client, the previous client file is overwritten, and the new client file must still be submitted to the depot. Note that it is possible for another user to have submitted yet another revision of the same file to the depot between the time `p4 resolve` completes and the time `p4 submit` is performed; in this case, it would be necessary to perform another resolve. This can be prevented by performing a `p4 lock` on the file before performing the resolve.

Using Flags with Resolve to Automatically Accept Particular Revisions

Five optional `p4 resolve` flags tell the command to work non-interactively. When these flags are used, particular revisions of the conflicting files are automatically accepted.

"p4 resolve" flag	Meaning
`-ay`	Automatically accept *`yours`*.
`-at`	Automatically accept *`theirs`*. Use this option with caution, as the file revision in the client workspace will be overwritten with no chance of recovery.
`-am`	Automatically accept the Perforce-recommended file revision: If *`theirs`* is identical to *`base`*, accept *`yours`*, if *`yours`* is identical to *`base`*, accept *`theirs`*, if *`yours`* and *`theirs`* are different from *`base`*, and there are no conflicts between *`yours`* and *`theirs`*, accept merge, otherwise, there are conflicts between *`yours`* and *`theirs`*, so skip this file.

"p4 resolve" flag	Meaning
`-af`	Accept the Perforce-recommended file revision, no matter what. If this option is used, the resulting file in the client should be edited to remove any difference markers.
`-as`	If *`theirs`* is identical to *`base`*, accept *`yours`*; if *`yours`* is identical to *`base`*, accept *`theirs`*; otherwise skip this file.

Example: *Automatically accepting particular revisions of conflicting files*

Ed has been editing the files in `doc/*.guide`*, and knows that some of them will require resolving. He types* `p4 sync doc/*.guide`*, and all of these files that conflict with files in the depot are scheduled for resolve.*

He then types `p4 resolve -am`*, and the merge files for all scheduled resolves are generated, and those merge files that contain no line set conflicts are written to his client workspace.*

He'll still need to manually resolve all the other conflicting files, but the amount of work he needs to do is substantially reduced.

Binary files and "p4 resolve"

If any of the three file revisions participating in the merge are binary instead of text, a three-way merge is not possible. Instead, `p4 resolve` performs a two-way merge: the two conflicting file versions are presented, and you can edit and choose between them.

Locking Files to Minimize File Conflicts

Once open, a file can be locked with `p4 lock` so that only the user who locked the file can submit the next revision of that file to the depot. Once the file is submitted, it is automatically unlocked. Locked files can also be unlocked manually by the locking user with `p4 unlock`.

The clear benefit of `p4 lock` is that once a file is locked, the user who locked it experiences no further conflicts on that file, and will not need to resolve the file. However, this comes at a price, as other users will not be able to submit the file until the file is unlocked, and will have to do their own resolves once they submit their revision. Under most circumstances, a user who locks a file is essentially saying to other users "I don't want to deal with any resolves; you do them."

There is an exception to this rule: Perforce also has a `+l` file type modifier to support exclusive-open. If you have a `+l` file open for edit, other users who attempt to edit the file will receive an error message.

The difference between `p4 lock` and `+l` is that `p4 lock` allows anyone to open a file for edit, but only the person who locked the file may submit it. By contrast, a file of type `+l` allows only one person to open the file for edit in the first place.

Preventing multiple resolves with p4 lock

Without file locking, there is no guarantee that the resolve process will ever end. The following scenario demonstrates the problem:

1. Ed opens file `foo` for edit.
2. Lisa opens the same file in her client for edit.
3. Ed and Lisa both edit their client workspace versions of `foo`.
4. Ed submits a changelist containing that file, and his submit succeeds.
5. Lisa submits a changelist with her version of the file; her submit fails because of file conflicts with the new depot's `foo`.
6. Lisa starts a resolve.
7. Ed edits and submits a new version of the same file.
8. Lisa finishes the resolve and attempts to submit; the submit fails and must now be merged with Ed's latest file.

   ...and so on

File locking can be used in conjunction with resolves to avoid this sort of headache. The sequence would be implemented as follows:

1. Before scheduling a resolve, lock the file.
2. Then sync the file, resolve the file, and submit the file.

As long as the locked file is open, new versions can't be submitted by other users until the resolved file is either submitted or unlocked.

Locked files appear in the output of `p4 opened` with an indication of `*locked*`. On UNIX, you can find all locked files you have open with the following command:

```
p4 opened | grep "*locked*"
```

This lists all open files you have locked with `p4 lock`.

Preventing multiple checkouts with +l files

If you know in advance that a certain file is only going to be worked on by a single user, you can use the `+l` (exclusive-open) filetype modifier to ensure that only one user at a time may work on the file.

You can change a file's type to exclusive-open by reopening it with the `+l` modifier. For instance:

```
p4 reopen -t binary+l file.gif
```

Although this prevents concurrent development, for some file types (usually binary files), merging and resolving may not be meaningful, and you may decide that you prefer to allow only one user to check these files out at a time.

If you find this style of work to be useful, you may want to ask your Perforce administrator to use the `p4 typemap` command (documented in the *Perforce Command Reference*) to ensure that *all* files matching a file specification (for instance, `//depot/.../*.gif` for all `.gif` files) are assigned type `+l` by default.

Resolves and Branching

Files in separate codelines can be integrated with `p4 resolve`; for details about resolving branched files, see Chapter 9, Branching.

Resolve Reporting

Four reporting commands are related to file conflict resolution: `p4 diff`, `p4 diff2`, `p4 sync -n`, and `p4 resolved`.

Command	Meaning
`p4 diff [filenames]`	Runs a diff program between the file revision currently in the client and the revision that was last gotten from the depot. If the file is not open for edit in the client, the two file revisions should be identical, so `p4 diff` fails. Comparison of the revisions can be forced with `p4 diff -f`, even when the file in the client is not open for edit Although `p4 diff` runs a diff routine internal to Perforce, this routine can be overridden by specifying an external diff in the `P4DIFF` environment variable.
`p4 diff2 file1 file2`	Runs `p4d`'s diff subroutine on any two Perforce depot files. The specified files can be any two file revisions, even revisions of entirely different files. The diff routine used by `p4d` cannot be overridden.
`p4 sync -n [filenames]`	Reports what the result of running `p4 sync` would be, without actually performing the sync. This is useful to see which files have conflicts and need to be resolved.
`p4 resolved`	Reports which files have been resolved but not yet submitted.

The reporting chapter (Chapter 11, Reporting and Data Mining) has a longer description of each of these commands. `p4 help` provides a complete listing of the many flags for these reporting commands.

Chapter 6 Perforce Basics: Miscellaneous Topics

The manual thus far has provided an introduction to the basic functionality provided by Perforce, and subsequent chapters cover the more advanced features. In between are a host of other, smaller facilities; this chapter covers these topics. Included here is information on the following:

- Changing your Perforce environment with the `P4CONFIG` environment variable,
- Using passwords to prevent impersonation by other users,
- Command-line flags common to all Perforce commands,
- How to work on files while not connected to a Perforce server,
- Refreshing the client workspace,
- Renaming files, and
- Recommendations for organization of files within the depot.

Reconfiguring the Perforce Environment with $P4CONFIG

Some Perforce users have multiple client workspaces, each of which may connect to a different port. There are three ways of changing your Perforce environment on the fly:

- Reset your environment or registry variables each time you want to move to a new workspace.
- Use command-line flags (discussed in the next section) to override the value of the environment variables `P4PORT`, `P4CLIENT`, and `P4USER`.
- Use the environment variable or registry variable `P4CONFIG` to point to a file containing a specification for the current Perforce environment.

`P4CONFIG` names a file (for example, `.p4env`) that is used to store variable settings. Whenever a Perforce command is executed, the current working directory and its parent directories are searched for a file with the name stored in `P4CONFIG`. If a file with that name is found, the values of `P4PORT`, `P4CLIENT`, and so on, are read from that file. If no file of the given name is found, the current values of the Perforce environment variables are used.

Each variable setting in the file stands alone on a line and must be in the form:

```
P4VAR=value
```

Values that can be stored in the P4CONFIG file are:

P4CLIENT	P4DIFF	P4EDITOR	P4USER	P4CHARSET
P4PORT	P4MERGE	P4PASSWD	P4HOST	P4LANGUAGE

Example: *Using* P4CONFIG *to automatically reconfigure the Perforce environment*

Ed often switches between two workspaces on the same machine. The first workspace is elmproj. *It has a client root of* /usr/edk/elm, *and connects to the Perforce server at* ida:1818. *The second workspace is called* graphicwork. *Its client root is* /usr/edk/other/graphics, *and it uses the Perforce server at* warhol:1666.

Ed sets the P4CONFIG *environment variable to* .p4settings. *He creates a file called* .p4settings *in* /usr/edk/elm *containing the following text:*

```
P4CLIENT=elmproj
P4PORT=ida:1818
```

He creates a second .p4settings *file in* /usr/edk/other/graphics. *It contains:*

```
P4PORT=warhol:1666
P4CLIENT=graphicwork
```

He always works within the directories where his files are located. Whenever Ed is anywhere beneath /usr/edk/other/graphics, *his Perforce client is* graphicwork, *and when he's in* /usr/edk/elmproj, *his client is* elmproj.

The values found in the file specified by P4CONFIG override any environment or registry variables that have been set. Command-line flags (discussed in the next section) override the values found in the P4CONFIG file.

P4CONFIG automates the process of changing the Perforce environment variables, as new settings take effect whenever you switch your current working directory from one client workspace directory to another. If you use multiple client workspaces, this is a very useful feature.

Perforce Passwords

By default, any user can assume the identity of any other Perforce user by setting the value of P4USER to a different username, by using the -u flag with the p4 command, or by setting the value of P4USER configuration within the file described by P4CONFIG. Although this makes it easy to perform tasks for other users when necessary, it can also lead to security problems.

To prevent another user from impersonating you within Perforce, use the p4 passwd command. No one, including the user who set the password, will be able to use any p4 command without providing the password to Perforce. You can provide this password to Perforce in one of three ways:

1. Set the value of the environment or registry variable `P4PASSWD` to the correct password.
2. Use the `-P password` flag between `p4` and the actual command when giving a Perforce command (for instance, `p4 -P eds_password submit`).
3. Set the value of `P4PASSWD` within the file described by `P4CONFIG`.

For security's sake, the password should always be set while logged onto the server.

Be careful when setting passwords. Once you have set your Perforce password with `p4 user`, there is no way for you to use Perforce if you forget their password and the value of `P4PASSWD` has not been properly set. If this happens, the Perforce superuser will have to reset or remove your password.

Note Passwords can also be created, changed, and deleted within the `p4 user` form.

If you need to have your password reset, contact your Perforce administrator. If you are a Perforce administrator, consult the *Perforce System Administrator's Guide* for information on resetting passwords and other common user-related tasks.

Command-Line Flags Common to All Perforce Commands

Some flags are available for use with all Perforce commands.

These flags are given between the system command `p4` and the command argument taken by `p4`. These flags are:

Flag	Meaning	Example
`-c clientname`	Runs the command on the specified client. Overrides the `P4CLIENT` environment variable.	`p4 -c joe edit //depot/foo` Opens file `foo` for editing under client workspace `joe`.
`-d directory`	Specifies the current directory, overriding the environment variable `PWD`.	`p4 -d ~elm/src edit foo bar` Opens files `foo` and `bar` for edit; these files are found relative to `~elm/src`.
`-p server_addr`	Gives the Perforce server's listening address, overriding `P4PORT`.	`p4 -p mama:1818 clients` Reports a list of clients on the server on host `mama`, port `1818`.

Flag	Meaning	Example
`-P` *`password`*	Supplies a Perforce password, overriding the value of `P4PASSWD`. Usually used in combination with the `-u` *`user`* flag.	`p4 -u ida -P idas_pw job` Create a new job as user `ida`, using `ida`'s Perforce password.
`-u` *`username`*	Specifies a Perforce user, overriding the `P4USER` environment variable. The user may run only those commands to which he or she has access.	`p4 -u bill user` Presents the `p4 user` form to edit the specification for user `bill`. The command will work without the `-P` flag only if `bill` has not set a Perforce password.
`-x` *`filename`*	Instructs `p4` to read arguments, one per line, from the named file.	See the *Working Detached* section, directly below.

All Perforce commands can take these flags, even commands for which these flag usages are clearly useless (for instance, `p4 -u bill -d /usr/joe help`). Other flags are available as well; these additional flags are command dependent. Please consult the *Perforce Command Reference* or use `p4 help` *`commandname`* to see the flags for each command.

Working Detached

Under normal circumstances, you work in your client workspace with a functioning network connection to a Perforce server. As you edit files, you announce your intentions to the server with `p4 edit`, and the server responds by noting the edit in the depot's metadata, and by unlocking the file in your client workspace. However, it is not always possible you to have a network connection to be present, and you may need a method for working entirely detached from the server.

The scheme is as follows:

- Work on files without giving Perforce commands. Instead, you use native OS commands to manually change the permissions on files, and then edit or delete the files.
- If you did not edit the files within the client workspace, copy them to the client workspace when the network connection is reestablished.
- Use `p4 diff` to find all files in your workspace that have changed or been added without Perforce's knowledge. Use the output from this command to bring the depot in sync with the client workspace.

Finding changed files with "p4 diff"

Use the `p4 diff` reporting command to compare a file in the client workspace with the corresponding file in the depot. The behavior of `p4 diff` can be modified with two flags:

"p4 diff" Variation	Meaning
`p4 diff -se`	Tells the names of unopened files that are present on the client, but whose contents are different than the files last taken by the client with `p4 sync`. These files are candidates for `p4 edit`.
`p4 diff -sd`	Reports the names of unopened files missing from the client. These files are candidates for `p4 delete`.

Note You can use `p4 edit` on any file, even files you don't want to edit; this command gives the local file write permissions, but does not otherwise alter it.

Using "p4 diff" to update the depot

The `p4 diff` variations described above can be used in combination with the `-x` flag to bring the state of the depot in sync with the changes made to the client workspace.

To open changed files for edit after working detached, use:

```
p4 diff -se > CHANGED_FILES
p4 -x CHANGED_FILES edit
```

To delete files from the depot that were removed from the client workspace, use:

```
p4 diff -sd > DEL_FILES
p4 -x DEL_FILES delete
```

As always, your edit and delete requests are stored in changelists, which Perforce does not process until you type `p4 submit`.

Refreshing files

The process of syncing a depot with a formerly detached client workspace has a converse: Perforce can get confused about the contents of a client workspace if you accidentally use the local OS file deletion command.

For example, suppose that you accidentally delete a client workspace file via the UNIX `rm` command, and that the file is one that you wanted to keep. Even after a submit, `p4 have` will still list the file as being present in the workspace.

In such a situation, you can use `p4 sync -f files` to bring the client workspace in sync with the files the depot *thinks* you have. This command is mostly a recovery tool for bringing the client workspace back into sync with the depot after accidentally removing or damaging files managed by Perforce.

Recommendations for Organizing the Depot

The default view brought up by `p4 client` maps the entire depot to the entire client workspace. If the client workspace is named `eds_elm`, the default view looks like this:

```
//depot/... //eds_elm/...
```

This is the easiest mapping, and can be used for the most simple Perforce depots, but mapping the entire depot to the workspace can lead to problems later on. Suppose your server currently stores files for only one project, but another project is added later: everyone who has a client workspace mapped as above will wind up receiving all the files from both projects into their workspaces. Additionally, the default view does not facilitate branch creation.

The safest way to organize the depot, even from the start, is to create one subdirectory per project within the depot. For example, if your company is working on three projects named `foo`, `bar`, and `zeus`, three subtrees might be created within the depot: `//depot/foo`, `//depot/bar`, and `//depot/zeus`. If Joe is working on the `foo` project, his mapping might look like this:

```
//depot/foo/...     //joe/...
```

And Sarah, who's working on the `bar` and `zeus` projects, might set up her client workspace as:

```
//depot/bar/...     //sarah/bar/...
//depot/zeus/...    //sarah/zeus/...
```

This sort of organization can be easily extended to as many projects and branches as are needed.

Another way of solving the same problem would be to have the Perforce system administrator create one depot for each project or branch. Please see the *Perforce System Administrators Guide* for details about setting up multiple depots.

> **Note** The depot is divided into subdirectories simply by setting up the proper mappings within the client views.

Renaming Files

Although Perforce doesn't have a rename command, you can rename files by using `p4 integrate` to copy the file from one location in the depot to another, deleting the file from the original location, and then submitting the changelist that includes the integrate and the delete. The process is as follows:

```
p4 integrate from_files to_files
p4 delete from_files
p4 submit
```

The *from_file* is moved to the directory and renamed according to the *to_file* specifier. For example, if *from_file* is `d1/foo` and *to_file* is `d2/bar`, then `foo` is moved to the `d2` directory, and is renamed `bar`. The *from_file* and *to_file* specifiers may include wildcards, as long as they are matched on both sides. Perforce `write` access (explained in the *Perforce System Administrator's Guide*) is needed on all the specified files.

Revision histories and renamed files

When you rename a file (or move it from one directory to another) with `p4 integrate`, you create the new file by creating an integration record that links it to its deleted predecessor.

As a result, if you run `p4 changes` *newfile*, you'll see only changes to *newfile*. Only changes that have taken place after the renaming will be listed:

```
$ p4 changes newfile.c
Change 4 on 2000/10/24 by user@client 'Latest bugfix'
Change 3 on 2000/10/23 by user@client 'Renamed file'
```

In order to see the *full* history of changes to the file (including changes made before the file was renamed or moved), you must specify the `-i` (include changes from integrations) flag to `p4 changes`, as follows:

```
$ p4 changes -i newfile.c
Change 4 on 2000/10/24 by user@client 'Latest bugfix'
Change 3 on 2000/10/23 by user@client 'Renamed file'
Change 2 on 2000/10/21 by user@client 'second version'
change 1 on 2000/10/20 by user@client 'initial check-in'
```

Specifying the `-i` flag tells `p4 changes` to trace back through the integration records and retrieve all change information, regardless of how many times the file (or the directory in which it lives) has been renamed or moved.

Chapter 7 Changelists

A Perforce *changelist* is a list of files, their revision numbers, and operations to be performed on these files. Commands such as `p4 add` *`filenames`* and `p4 edit` *`filenames`* include the affected files in a changelist; the depot is not actually altered until the changelist is submitted with `p4 submit`.

When a changelist is submitted to the depot, the depot is updated *atomically*: either all of the files in the changelist are updated in the depot, or none of them are. This grouping of files as a single unit guarantees that code alterations spanning multiple files are updated in the depot simultaneously. To reflect the atomic nature of changelist submissions, submission of a changelist is sometimes called an *atomic change transaction*.

Perforce attempts to make changelist usage as transparent as possible: in the normal case, Perforce commands such as `p4 edit` add the affected files to the *default changelist*, and `p4 submit` sends the default changelist to the server for processing. However, there are two sets of circumstances that would require the user to understand and manipulate non-default changelists:

- Sometimes a user wants to split files into separate groups for submission.

  For example, suppose a user is fixing two bugs, each of which spans a separate set of files. Rather than submit the fixes to both bugs in a single changelist, the user might elect to create one changelist for the files that fix the first bug, and another changelist for the files that fix the second bug. Each changelist would be submitted to the depot via separate `p4 submit` operations.

- Under certain circumstances, the `p4 submit` command can fail.

  For example, if one user has a file locked and another user submits a changelist that contains that file, the submit fails. When a submit of the default changelist fails, the changelist is assigned a number, is no longer the default changelist, and must be referred to by its number.

In the above circumstances, the user must understand how to work with *numbered changelists*.

Working with the Default Changelist

Note The material in this subsection has already been presented in a slightly different form in earlier chapters. It is presented again here to provide a complete discussion of changelists.

A changelist is a list of files, revision numbers of those files, and operations to be performed on those files. For example, a single changelist might contain the following:

```
/doc/elm-help.1       revision 3    edit
/utils/elmalias.c     revision 2    delete
```

Each of the files in the changelist is said to be *open* within the client workspace: the first of the files above was opened for edit with `p4 edit`, and the second was opened for deletion with `p4 delete`.

The files in the changelist are updated within the depot with `p4 submit`, which sends the changelist to the server; the server processes the files contained in the changelist and alters the depot accordingly.

The commands that add or remove files from changelists are:

`p4 add`	`p4 delete`	`p4 edit`
`p4 integrate`	`p4 reopen`	`p4 revert`

By default, these commands, and `p4 submit`, act on the default changelist. For example, if you type `p4 add` *`filename`*, this file is added to the default changelist. When you type `p4 submit`, the default changelist is submitted.

When you type `p4 submit`, a change form is displayed, containing the files in the default changelist. Any file can be deleted from this list; when a file is deleted, it is moved to the next default changelist, and will appear again the next time you type `p4 submit`. A changelist must contain a user-entered description, which describes the nature of the changes being made.

`p4 submit` can take an optional, single file pattern as an argument. In this case, only those files in the default changelist that match the file pattern are included in the submitted changelist. Since the `p4d` server program must receive this file pattern as a single argument, make sure to escape the `*` wildcard if it is used.

When the user quits from the `p4 submit` editor, the changelist is submitted to the server and the server attempts to update the files in the depot. If there are no problems, the changelist is assigned a sequential number, and its status changes from `new` or `pending` to `submitted`. Once a changelist has been submitted, it becomes a permanent part of the depot's metadata, and is unchangeable except by Perforce administrators.

Creating Numbered Changelists Manually

A user can create a changelist in advance of submission with `p4 change`. This command brings up the same form that you see during `p4 submit`.

All files in the default changelist are moved to this new changelist. When you quit from the form, the changelist is assigned the next changelist number in sequence, and this changelist must be subsequently referred to by this change number. Files can be deleted from the changelist by editing the form; files deleted from this changelist are moved to the next default changelist. The status for a changelist created by this method is `pending` until you submit the files in the changelist.

Any single client file can be included in only one pending changelist.

Working With Numbered Changelists

You can use commands such as `p4 edit` *`filename`*, which by default adds the files to the default changelist, to append a file to a pending numbered changelist with the `-c` *`changenum`* flag. For example, to edit a file and submit it in changelist number 4, use `p4 edit -c 4` *`filename`*.

You can move files from one changelist to another with `p4 reopen -c` *`changenum filenames`*, where *`changenum`* is the number of the moving-to changelist. If you are moving files to the default changelist, use `p4 reopen -c default` *`filenames`*.

Example: *Working with multiple changelists.*

Ed is working on two bug fixes simultaneously. One of the bugs involves mail filtering and requires updates of files in the `filter` *subdirectory; the other problem is in the aliasing system, and requires an update of* `utils/elmalias.c`.

Ed wants to update each bug separately in the depot; this will allow him to refer to one bug fix by one change number and the other bug fix by another change number. He's already started fixing both bugs, and has opened some of the affected files for edit. He types `p4 change`, *and sees*

```
Change: new
Client: eds_elm
User:   edk
Status: new
Description:
        <enter description here>
Files:
        //depot/elm_proj/filter/filter.c   # edit
        //depot/elm_proj/filter/lock.c     # edit
        //depot/elm_proj/utils/elmalias.c  # edit
```

Ed wants to use this changelist to submit only the fix to the filter problems. He changes the form, deleting the last file revision from the file list. When he's done, the form looks like this:

```
Change: new
Client: eds_elm
User:   edk
Status: new
Description:
        Fixes filtering problems
Files:
        //depot/elm_proj/filter/filter.c   # edit
        //depot/elm_proj/filter/lock.c     # edit
```

When he quits from the editor, he's told

```
Change 29 created with 2 open file(s).
```

The file that he removed from the list, `utils/elmalias.c`*, is now found in the default changelist. He could include that file in another numbered changelist, but decides to leave it where it is.*

He fixes both bugs at his leisure. He realizes that the filter problem requires updates to another file: `filter/lock.c`*. He opens this file for edit with* `p4 edit -c 29 filter/lock.c`*; opening the file with the* `-c 29` *flag puts the file in changelist 29, which he created above. (If the file had already been open for edit in the default changelist, he could have moved it to changelist 29 with* `p4 reopen -c 29 filter/lock.c`*).*

Ed finishes fixing the aliasing bug, and, since the affected files are in the default changelist, he submits the changelist with a straightforward `p4 submit`*. He'll finish fixing the filtering bug later.*

Automatic Creation and Renumbering of Changelists

When submit of the default changelist fails, the changelist is assigned a number

Submits of changelists occasionally fail. This can happen for a number of reasons:

- A file in the changelist has been locked by another user with `p4 lock`.
- The client workspace no longer contains a file included in the changelist.
- There is a server error, such as not enough disk space.
- The user was not editing the head revision of a particular file. The following sequence shows an example of how this can occur:
  - User A types `p4 edit //depot/foo`,
  - User B types `p4 edit //depot/foo`,

- User B submits her default changelist, and
- User A submits his default changelist.

User A's submit is rejected, since the file revision of `foo` that he edited is no longer the head revision of that file.

If any file in a changelist is rejected for any reason, the entire changelist is backed out, and none of the files in the changelist are updated in the depot. If the submitted changelist was the default changelist, Perforce assigns the changelist the next change number in sequence, and this change number must be used from this point on to refer to the changelist. Perforce also locks the files to prevent others from changing them while the user resolves the reason for the failed submit.

If the submit failed because the client-owned revision of the file is not the head revision, a *merge* must be performed before the changelist will be accepted. (File merging is described in Chapter 5, Perforce Basics: Resolving File Conflicts).

Perforce May Renumber a Changelist upon Submission

The change numbers of submitted changelists always reflect the order in which the changelists were submitted. Thus, when a changelist is submitted, it may be renumbered.

Example: *Automatic renumbering of changelists*

Ed has finished fixing the filtering bug that he's been using changelist 29 for. Since he created that changelist, he's submitted another changelist (change 30), and two other users have submitted changelists. Ed submits change 29 with `p4 submit -c 29`*, and is told:*

```
Change 29 renamed change 33 and submitted.
```

Deleting Changelists

To remove a pending changelist that has no files or jobs associated with it, use `p4 change -d` *`changenum`*. Pending changelists that contain open files or jobs must have the files and jobs removed from them before they can be deleted: use `p4 reopen` to move files to another changelist, `p4 revert` to remove files from the changelist (and revert them back to their old versions), or `p4 fix -d` to remove jobs from the changelist.

Changelists that have already been submitted can be deleted by a Perforce administrator only under very specific circumstances. Please see the *Perforce System Administrator's Guide* for more information.

Changelist Reporting

The two reporting commands associated with changelists are `p4 changes` and `p4 describe`. The former is used to view lists of changelists with short descriptions, while the latter prints verbose information for a single changelist.

Command	Meaning
`p4 changes`	Displays a list of all pending and submitted changelists, one line per changelist, and an abbreviated description.
`p4 changes -m` *`count`*	Limits the number of changelists reported on to the last *`count`* changelists.
`p4 changes -s` *`status`*	Limits the list to those changelists with a particular status; for example, `p4 changes -s submitted` will list only already submitted changelists.
`p4 changes -u` *`user`*	Limits the list to those changelists submitted by a particular user.
`p4 changes -c` *`workspace`*	Limits the list to those changelists submitted from a particular client workspace.
`p4 describe` *`changenum`*	Displays full information about a single changelist. If the changelist has already been submitted, the report includes a list of affected files and the diffs of these files. (You can use the `-s` flag to exclude the file diffs.)

Chapter 8 Labels

A Perforce *label* is a user-determined list of files and revisions. The label can later be used to reproduce the state of these files within a client workspace.

Labels provide a method of naming important combinations of file revisions for later reference. For example, the file revisions that comprise a particular release of your software might be given the label `release2.0.1`. At a later time, you can retrieve all the files in that label into a client workspace with a single command.

Create a label when:

- You want to keep track of all the file revisions contained in a particular release of the software,
- There exists a particular set of file revisions that you want to give to other users, or
- You have a set of file revisions that you want to branch from, but you don't want to perform the branch yet. In this case, you would create a label for the file revisions that will form the base of the branch.

Why Not Just Use Changelist Numbers?

Labels share certain important characteristics with change numbers, as both refer to particular file sets, and both act as handles to refer to all the files in the set. But labels have some important advantages over change numbers:

- The file revisions referenced by a particular label can come from different changelists.
- A change number refers to the state of all the files in the depot at the time the changelist was submitted, while a label can refer to any arbitrary set of files and revisions.
- The files and revisions referenced by a label can be arbitrarily changed at any point in the label's existence.
- Changelists are always referred to by Perforce-assigned numbers, while labels are named by the user.

Creating a Label

You can create a label with `p4 label` *`labelname`*. This command brings up a form similar to the `p4 client` form. Like clients, labels have associated views; the label view limits which files can be referenced by the label. Once you have created the label, use `p4 labelsync` to load the label with file references.

Label names share the same namespace as clients, branches, and depots. Thus, a label name cannot be the same as any existing client, branch, or depot name.

Example: *Creating a Label*

Ed has finished the first version of filtering in `elm`*. He wants to create a label that references only those files in the* `filter` *and* `hdrs` *subdirectories. He wants to name the label* `filters.1`*. He types* `p4 label filters.1` *and fills in the resulting form as follows:*

```
Label:  filters.1
Owner:  edk
Description:
        Created by edk.
Options:        unlocked
View:
        //depot/elm_proj/filter/...
        //depot/elm_proj/hdrs/...
```

When he quits from the editor, the label is created.

Before following this example further, it's worth stopping for a moment to examine exactly what has and hasn't been accomplished. So far, a label called `filters.1` has been created. It can contain files only from the depot's `elm_proj filter` and `hdrs` subdirectories. But the label `filters.1` is empty; it contains no file references. It will be loaded with its file references with `p4 labelsync`.

The `View:` field limits the files that are included in the label. These files must be specified by their location in the depot; this view differs from other views in that only the depot side of the view is specified.

The `locked` / `unlocked` option in the `Options:` field can prevent `p4 labelsync` from overwriting previously synced labels (this is described in "Preventing Accidental Overwrites of a Label's Contents" on page 94).

Adding and Changing Files Listed in a Label

Once you've created a label, you can include references to files within it by using `p4 labelsync`. This command's syntax is

```
p4 labelsync [-a -d -n] -l labelname [filename...]
```

The rules followed by `labelsync` to include files in a label are as follows:

1. You must be the owner of the label in order to use `p4 labelsync` on it, and the label must be `unlocked`.

If you are not the owner of a label, you may (assuming you have sufficient permissions) make yourself the owner by running:

```
p4 label labelname
```

and changing the `Owner:` field to your userid in the `p4 label` form. Similarly, you may unlock a label by setting the `Options:` field (also in the `p4 label` form) to `unlocked`.

2. All files listed in a label must be contained in the label view specified in the `p4 label` form. Any files or directories that are not mapped through the label view are ignored by `p4 labelsync`. All the following rules assume this, without further mention.

3. When `p4 labelsync` is used to include a particular file in a label's file list, the file is added to the label if it is not already included in the label. If a different revision of the file is already included in the label's file list, it is replaced with the newly specified revision. Only one revision of any file is ever included in a label's file list.

4. If `labelsync` is called with no filename arguments, as in:

   ```
   p4 labelsync -l labelname
   ```

   then all the files mapped by the label view are listed in the label. The revisions added to the label are those last synced into the client; these revisions can be seen in the `p4 have` list. Calling `p4 labelsync` this way replaces *all* existing file references with the new ones.

Example: *Including file references in a label with* `p4 labelsync`.

Ed has created a label called `filters.1` *as specified above, and now he wants to load the* `filters.1` *label with the proper file revisions. He types:*

```
p4 labelsync -l filters.1
```

and sees the following:

```
//depot/elm_proj/filter/Makefile.SH#20 - added
//depot/elm_proj/filter/actions.c#25 - added
<etc.>
```

Perforce adds those file revisions to the label that are those contained in the intersection of the label view and the current client `have` *list.*

5. When you call `p4 labelsync` with file pattern arguments, and the arguments contain no revision specifications, the head revisions of these files are included in the label's file list.

Example: *Changing file references in a label with* `p4 labelsync`.

After performing the above `labelsync` *command, Ed finds that the file* `filter/filter.c` *is buggy. He fixes it, submits the new version, and wants to replace the old revision of this file in the label* `filters.1` *with the new revision. From the* `filter` *subdirectory, he types*

```
p4 labelsync -l filters.1 filter/filter.c
```

and sees

```
//depot/elm_proj/filter/filter.c#15 - updated
```

The head revision of `filter.c` *replaces the revision that had been previously included in the label.*

6. If you call `p4 labelsync` with file pattern arguments containing revision specifications, these file revisions are included in the label's file list.

Example: *Including a different file revision in a label*

Ed realizes that the version of `filter/audit.c` *contained in his label* `filters.1` *is not the version he wants to include in the label. He'd prefer to include revision 12 of that file. From the main* `Elm` *directory, he types*

```
p4 labelsync -l filters.1 filter/audit.c#12
```

and sees:

```
/depot/elm_proj/filter/audit.c#12 - updated
```

This revision of `audit.c` *replaces the revision that had been previously included in the label.*

Previewing labelsync's results

The results of `p4 labelsync` can be previewed with `p4 labelsync -n`. This lists the files that would be added, deleted, or replaced in the label without actually performing the operation.

Preventing Accidental Overwrites of a Label's Contents

Since `p4 labelsync` with no file arguments overwrites all the files that are listed in the label, it is possible to accidentally lose the information that a label is meant to contain. To prevent this, call `p4 label` *`labelname`* and set the value of the `Options:` field to `locked`. It will be impossible to call `p4 labelsync` on that label until the label is subsequently `unlocked`.

Retrieving a Label's Contents into a Client Workspace

To retrieve all the files listed in a label into a client workspace, use `p4 sync` *`files@labelname`*. Rather than simply adding the files to the client workspace, this command matches the state of the client workspace to the state of the label. Thus, files in the client workspace that aren't in the label may be deleted from the client workspace.

Example: *Retrieving files into a client workspace from a label.*

Lisa wants to retrieve all the files listed in Ed's `filters.1` *label into her client workspace. She can type:*

```
p4 sync //depot/...@filters.1
```

or even:

```
p4 sync @filters.1
```

Instead, she's interested in seeing only the header files from that label, so she types:

```
p4 sync //depot/elm_proj/hdrs/*@filters.1
```

and sees:

```
//depot/elm_proj/hdrs/curses.h#1 - added as /usr/lisag/elm/hdrs/curses.h
//depot/elm_proj/hdrs/defs.h#1 - added as /usr/lisag/elm/hdrs/defs.h
//depot/elm_proj/hdrs/test.h#3 - deleted as /usr/lisag/elm/hdrs/test.h
<etc>
```

All the files in the subdirectory `hdrs` *that are within the intersection of Ed's* `filters.1` *label and Lisa's client view are retrieved into her workspace. But there is another effect as well: files that are not in the intersection of the label's contents and* `//depot/elm_proj/hdrs/*` *are deleted from her workspace.*

If `p4 sync` *`@labelname`* is called with no file parameters, all files in the client that are not in the label will be deleted from the client. If this command is called with file arguments, as in `p4 sync` *`files@labelname`*, then the client workspace at the intersection of the depot, the client workspace view, and the file arguments will be updated to match the contents of the depot at that intersection.

Deleting Labels

You can delete a label with:

```
p4 label -d labelname
```

You can delete files from labels with:

```
p4 labelsync -d -l labelname filepattern
```

To delete all the files from the label's file list, but leave the empty label in the Perforce database, use:

```
p4 labelsync -d -l labelname
```

Label Reporting

The commands that provide reports on labels are:

Command	Description
`p4 labels`	Report the names, dates, and descriptions of all labels known to the server
`p4 labels file#revrange`	Report the names, dates, and descriptions of all labels that include the specified revision range of *file*.
`p4 files @labelname`	Lists all files and revisions contained in the given label.
`p4 sync -n @labelname`	Shows what `p4 sync` would do when retrieving files from a particular label into your client workspace, without actually performing the sync.

Chapter 9 Branching

Perforce's *Inter-File Branching*™ mechanism allows any set of files to be copied within the depot, and allows changes made to one set of files to be copied, or *integrated*, into another. By default, the new file set (or *codeline*) evolves separately from the original files, but changes in either codeline can be propagated to the other with the `p4 integrate` command.

What is Branching?

Branching is a method of keeping two or more sets of similar (but not identical) files synchronized. Most software configuration management systems have some form of branching; we believe that Perforce's mechanism is unique because it mimics the style in which users create their own file copies when no branching mechanism is available.

Suppose for a moment that you're writing a program and are not using an SCM system. You're ready to release your program: what would you do with your code? Chances are that you'd copy all your files to a new location. One of your file sets would become your release codeline, and bug fixes to the release would be made to that file set; your other file files would be your development file set, and new functionality to the code would be added to these files.

What would you do when you find a bug that's shared by both file sets? You'd fix it in one file set, and then copy the edits that you made into the other file set.

The only difference between this homegrown method of branching and Perforce's branching methodology is that Perforce *manages the file copying and edit propagation for you*. In Perforce's terminology, copying the files is called *making a branch*. Each file set is known as a *codeline*, and copying an edit from one file set to the other is called *integration*. The entire process is called *branching*.

When to Create a Branch

Create a branch when two sets of code files have different rules governing when code can be submitted, or whenever a set of code files needs to evolve along different paths. For example:

- The members of the development group want to submit code to the depot whenever their code changes, whether or not it compiles, but the release engineers don't want code to be submitted until it's been debugged, verified, and signed off on.

They would branch the release codeline from the development codeline. When the development codeline is ready, it is integrated into the release codeline. Afterwards, patches and bug fixes are made in the release code, and at some point in the future, integrated back into the development code.

- A company is writing a driver for a new multi-platform printer. It has written a UNIX device driver, and is now going to begin work on a Macintosh driver using the UNIX code as their starting point.

  The developers create a branch from the existing UNIX code, and now have two copies of the same code. These two codelines can then evolve separately. If bugs are found in either codeline, bug fixes can be propagated from one codeline to the other with the Perforce `p4 integrate` command.

One basic strategy is to develop code in `//depot/main/` and create branches for releases (for example, `//depot/rel1.1/`). Make bug fixes that affect both codelines in `//depot/main/` and integrate them into the release codelines. Make release-specific bug fixes in the release branch and, if required, integrate them back into the `//depot/main/` codeline.

Perforce's Branching Mechanisms: Introduction

Perforce provides two mechanisms for branching. One method requires no special setup, but requires the user to manually track the mappings between the two sets of files. The second method remembers the mappings between the two file sets, but requires some additional work to set up.

In the first method, the user specifies both the files that changes are being copied from and the files that the changes are being copied into. The command looks like this:

```
p4 integrate fromfiles tofiles
```

In the second method, Perforce stores a mapping that describes which set of files get branched to other files, and this mapping, or branch specification, is given a name. The command the user runs to copy changes from one set of files to the other looks like this:

```
p4 integrate -b branchname [tofiles]
```

These methods are described in the following two sections.

Branching and Merging, Method 1: Branching with File Specifications

Use `p4 integrate` *`fromfiles tofiles`* to propagate changes from one set of files (the *source files*) to another set of files (the *target files*). The target files need to be contained within the current client workspace view. The source files do not need to be, so long as the source files are specified in depot syntax. If the target files do not yet exist, the entire contents of the source files are copied to the target files. If the target files have already been created, changes can be propagated from one set of files to the other with `p4 resolve`. In both cases, `p4 submit` must be used to store the new file changes in the depot. Examples and further details are provided below.

Creating branched files

To create a copy of a file that will be tracked for branching, use the following procedure:

1. Determine where you want the copied (or *branched*) file(s) to reside within the depot and within the client workspace. Add the corresponding mapping specification to your client view.
2. Run `p4 integrate` *`fromfiles tofiles`*. The source files are copied from the server to target files in the client workspace.
3. Run `p4 submit`. The new files are created within the depot, and are now available for general use.

Example: *Creating a branched file.*

Version 2.0 of Elm has just been released, and work on version 3.0 is about to commence. Work on the current development release always proceeds in `//depot/elm_proj/...`*, and it is determined that maintenance of version 2.0 will take place in* `//depot/elm_r2.0/...` *The files in* `//depot/elm_proj/...` *need to be branched into* `//depot/elm_r2.0/...`*, so Ed does the following:*

He decides that he'll want to work on the new `//depot/elm_r2.0/...` *files within his client workspace at* `/usr/edk/elm_proj/r2.0`*. He uses* `p4 client` *to add the following mapping to his client view:*

```
//depot/elm_r2.0/... //eds_elm/r2.0/...
```

He runs

```
p4 integrate //depot/elm_proj/... //depot/elm_r2.0/...
```

which copies all the files under `//depot/elm_proj/...` *to* `//eds_elm/v2.0` *in his client workspace. Finally, he runs* `p4 submit`*, which adds the new branched files to the depot.*

Why not just copy the files?

Although it is possible to accomplish everything that has been done thus far by copying the files within the client workspace and using `p4 add` to add the files to the depot, when you use `p4 integrate`, Perforce is able to track the connections between related files in an *integration record*, allowing easy propagation of changes between one set of files and another.

Propagating changes between branched files

After a file has been branched from another with `p4 integrate`, Perforce can track changes that have been made in either set of files and merge them using `p4 resolve` into the corresponding branch files. (You'll find a general discussion of the resolve process in Chapter 5, Perforce Basics: Resolving File Conflicts. File resolution with branching is discussed in "How Integrate Works" on page 106).

The procedure is as follows:

1. Run `p4 integrate` *`fromfiles tofiles`* to tell Perforce that changes in the source files need to be propagated to the target files.
2. Use `p4 resolve` to copy changes from the source files to the target files. The changes are made to the target files in the client workspace.
3. Run `p4 submit` to store the changed target files in the depot.

Example: *Propagating changes between branched files.*

Ed has created a release 2.0 branch of the Elm source files as above, and has fixed a bug in the original codeline's `src/elm.c` *file. He wants to merge the same bug fix to the release 2.0 codeline. From his home directory, Ed types*

```
p4 integrate elm_proj/src/elm.c //depot/elm_r2.0/src/elm.c
```

and sees

```
//depot/elm_r2.0/src/elm.c#1 - integrate from //depot/elm_proj/src/elm.c#9
```

The file has been scheduled for resolve. He types `p4 resolve`*, and the standard merge dialog appears on his screen.*

```
/usr/edk/elm_r2.0/src/elm.c - merging //depot/elm_proj/src/elm.c#2
Diff chunks: 0 yours + 1 theirs + 0 both + 0 conflicting
Accept(a) Edit(e) Diff(d) Merge (m) Skip(s) Help(?) [at]:
```

He resolves the conflict with the standard use of `p4 resolve`*. When he's done, the result file overwrites the file in his branched client, and it still must be submitted to the depot.*

There is one fundamental difference between resolving conflicts in two revisions of the same file, and resolving conflicts between the same file in two different codelines. The difference is that Perforce will detect conflicts between two revisions of the same file and

then schedule a resolve, but there are *always* differences between two versions of the same file in two different codelines, and these differences usually don't need to be resolved manually. (In these cases, a `p4 resolve -as` or `p4 resolve -am` to accept the Perforce-recommended revision is usually sufficient. See "Using Flags with Resolve to Automatically Accept Particular Revisions" on page 71 for details.)

In their day-to-day use, there is *no* difference between branched files and non-branched files. The standard Perforce commands like `sync`, `edit`, `delete`, `submit`, and so on. are used with all files, and evolution of both codelines proceeds separately. When changes to one codeline need to be propagated to another, you must tell Perforce to do this with `p4 integrate`, but if the codelines evolve separately, and changes never need to be propagated, you'll never need to integrate or resolve the files in the two codelines.

Propagating changes from branched files to the original files

A change can be propagated in the reverse direction, from branched files to the original files, by supplying the branched files as the source files, and the original files as the target files.

Example: *Propagating changes from branched files to the original files.*

Ed wants to integrate some changes in //depot/elm_r2.0/src/screen.c file to the original version of the same file. He types

```
p4 integrate //depot/elm_r2.0/src/screen.c //depot/elm_proj/src/screen.c
```

and then runs `p4 resolve`. *The changes in the branched file can now be merged into his source file.*

Branching and Merging, Method 2: Branching with Branch Specifications

To map a set of source files to target files, you can create a *branch specification* and use it as an argument to `p4 integrate`. To create and use a branch specification, do the following:

1. Use `p4 branch` *`branchname`* to create a view that indicates which target files map to which source files.
2. Make sure that the new files and directories are included in the `p4 client` view of the client workspace that will hold the new files.
3. Use `p4 integrate -b` *`branchname`* to create the new files.
4. To propagate changes from source files to target files, use `p4 integrate -b` *`branchname`* [*`tofiles`*]. Perforce uses the branch specification to determine which files the merged changes come from

5. Use `p4 submit` to submit the changes to the target files to the depot.

The following example demonstrates the same branching that was performed in the example above, this time using a branch specification.

Example: *Creating a branch.*

Version 2.0 of Elm has just been released, and work on version 3.0 is about to commence. Work on the current development release always proceeds in `//depot/elm_proj/...`*, and it is determined that maintenance of version 2.0 will take place in* `//depot/elm_r2.0/...` *The files in* `//depot/elm_proj/...` *need to be branched into* `//depot/elm_r2.0/...`*, so Ed does the following:*

Ed creates a branch specification called `elm2.0` *by typing* `p4 branch elm2.0`*. The following form is displayed:*

```
Branch: elm2.0
Date:   1997/05/25 17:43:28
Owner:  edk
Description:
        Created by edk.
View:
        //depot/... //depot/...
```

The view maps the original codeline's files (on the left) to branched files (on the right). Ed changes the `View:` *and* `Description:` *fields as follows:*

```
Branch: elm2.0
Date:   1997/05/25 17:43:28
Owner:  edk
Description:
        Elm release 2.0 maintenance codeline
View:
        //depot/elm_proj/... //depot/elm_r2.0/...
```

Ed wants to work on the new `//depot/elm_r2.0/...` *files within his client workspace at* `/usr/edk/elm_proj/r2.0`*. He uses* `p4 client` *to add the following mapping to his client view:* `//depot/elm_r2.0/... //eds_elm/r2.0/...`

He runs `p4 integrate -b elm2.0`*, which copies all the files under* `//depot/elm_proj/...` *to* `//eds_elm/r2.0/...` *in his client workspace; then he runs* `p4 submit`*, which adds the newly branched files to the depot.*

Once the branch has been created and the files have been copied into the branched codeline, changes can be propagated from the source files to the target files with `p4 integrate -b` *branchname*.

Example: *Propagating changes to files with* `p4 integrate`*.*

A bug has been fixed in the original codeline's `src/elm.c` *file. Ed wants to propagate the same bug fix to the branched codeline he's been working on. He types*

```
p4 integrate -b elm2.0 ~edk/elm_r2.0/src/elm.c
```

and sees:

```
//depot/elm_r2.0/src/elm.c#1 - integrate from //depot/elm_proj/src/elm.c#9
```

The file has been scheduled for resolve. He types `p4 resolve`*, and the standard merge dialog appears on his screen.*

```
/usr/edk/elm_r2.0/src/elm.c - merging //depot/elm_proj/src/elm.c#9
Diff chunks: 0 yours + 1 theirs + 0 both + 0 conflicting
Accept(a) Edit(e) Diff(d) Merge (m) Skip(s) Help(?) [at]:
```

He resolves the conflict with the standard use of `p4 resolve`*. When he's done, the result file overwrites the file in his branched client, and it still must be submitted to the depot.*

Branch Specification Usage Notes

- *Creating or altering a branch specification has absolutely no immediate effect on any files within the depot or client workspace.* The branch specification merely specifies which files are affected by subsequent `p4 integrate` commands.
- Branch specifications may contain multiple mappings, just as client views can. For example, the following branch specification branches the `Elm 2.0` source code and documents to two separate locations within the depot:

```
Branch: elm2.0
Date:   1997/05/25 17:43:28
Owner:  edk
Description:
        Elm release 2.0 maintenance codeline
View:
        //depot/elm_proj/src/...  //depot/elm_r2.0/src/...
        //depot/elm_proj/docs/... //depot/docs/2.0/...
```

- Exclusionary mappings can be used within branch specifications.
- To reverse the direction of an integration that uses a branch specification, use the `-r` flag.

Integration Usage Notes

- `p4 integrate` only acts on files that are the intersection of target files in the branch view and the client view. If file patterns are given on the command line, `integrate` further limits its actions to files matching the patterns. The source files supplied as arguments to `integrate` need not be in the client view.

- The basic syntax of the integrate command when using a branch specification is:

  ```
  p4 integrate -b branchname [tofiles]
  ```

  If you omit the *tofiles* argument, all the files in the branch are affected.

- The direction of integration through a branch specification may be reversed with the `-r` flag. For example, to integrate changes from a branched file to the original source file, use `p4 integrate -b branchname -r [tofiles]`

- The `p4 integrate` command, like `p4 add`, `p4 edit`, and `p4 delete`, does not actually affect the depot immediately; instead, it adds the affected files to a changelist, which must be submitted with `p4 submit`. This keeps the `integrate` operation atomic: either all the named files are affected at once, or none of them are.

- The actual action performed by `p4 integrate` is determined by particular properties of the source files and the target files:

  If the target file doesn't exist, the source file is copied to *target*, *target* is opened for `branch`, and Perforce begins tracking the integration history between the two files. The next integration of the two files will treat this revision of *source* as *base*.

  If the target file exists, and was originally branched from the source file with `p4 integrate`, then a three-way merge is scheduled between *target* and *source*. The base revision is the previously integrated revision of *source*.

  If the target file exists, but was not branched from the source, then these two file revisions did not begin their lives at a common, older file revision, so there can be no *base* file, and `p4 integrate` rejects the integration. This is referred to as a *baseless merge*. To force the integration, use the `-i` flag; `p4 integrate` will use the first revision of *source* as *base*. (Actually, `p4 integrate` uses the most recent revision of *source* that was added to the depot as *base*. Since most files are only opened for `add` once, this will almost always be the first revision of *source*.)

  Note In previous versions of Perforce (99.1 and earlier), integration of a target that was not originally branched from the source would schedule a two-way merge, in which the only resolve choices were `accept yours` and `accept theirs`. As of Perforce 99.2, it is no longer possible to perform a two-way merge of a text file (even when possible, it was never desirable).

- By default, a file that has been newly created in a client workspace by `p4 integrate` cannot be edited before its first submission. To make a newly-branched file available for editing before submission, perform a `p4 edit` of the file after the resolve process is complete.

- To run the `p4 integrate` command, you need Perforce `write` access on the target files, and `read` access on the source files. (See the *Perforce System Administrator's Guide* for information on Perforce protections).

Deleting Branches

To delete a branch, use

```
p4 branch -d branchname
```

Deleting a branch deletes only the branch specification, making the branch specification inaccessible from any subsequent `p4 integrate` commands. The files themselves can still be integrated with `p4 integrate` *`fromfiles tofiles`*, and the branch specification can always be redefined. If the files in the branched codeline are to be removed, they must be deleted with `p4 delete`.

Advanced Integration Functions

Perforce's branching mechanism also allows integration of specific file revisions, the re-integration and re-resolving of already integrated code, and merging of two files that were previously not related.

Integrating specific file revisions

By default, the `integrate` command integrates into the target all the revisions of the source since the last source revision that `integrate` was performed on. A revision range can be specified when integrating; this prevents unwanted revisions from having to be manually deleted from the merge while editing. In this case, the revision used as *`base`* is the first revision below the specified revision range.

The argument to `p4 integrate` is the target, the file revision specifier is applied to the source.

Example: *Integrating Specific File Revisions.*

Ed has made two bug fixes to his file `src/init.c`*, and Kurt wants to integrate the change into his branched version, which is called* `newinit.c`*. Unfortunately,* `init.c` *has gone through 20 revisions, and Kurt doesn't want to have to delete all the extra code from all 20 revisions while resolving.*

Kurt knows that the bug fixes he wants were made to file revisions submitted in changelist 30. From the directory containing his `newinit.c` *file in his branched workspace, he types*

```
p4 integrate -b elm_r1 newinit.c@30,@30
```

The target file is given as an argument, but the file revisions are applied to the source. When Kurt runs `p4 resolve`*, only the revision of Ed's file that was submitted in changelist 30 is*

scheduled for resolve. That is, Kurt only sees the changes that Ed made to `init.c` *in changelist 30. The file revision that was present in the depot at changelist 29 is used as* `base`*.*

Re-integrating and re-resolving files

After a revision of a source file has been integrated into a target, that revision is usually skipped in subsequent integrations with the same target. If all the revisions of a source have been integrated into a particular target, `p4 integrate` returns the error message `All revisions already integrated`. To force the integration of already-integrated files, specify the `-f` flag to `p4 integrate`.

Similarly, a target that has been resolved but not (yet) submitted can be re-resolved by specifying the `-f` flag to `p4 resolve`, which forces re-resolution of already resolved files. When this flag is used, the original client target file has been replaced with the result file by the original resolve process; when you re-resolve, *`yours`* is the new client file, the result of the original resolve.

How Integrate Works

The following sections describe the mechanism behind the integration process.

The *yours*, *theirs*, and *base* files

The following table explains the terminology `yours`, `theirs`, and `base` files.

Term	Meaning
`yours`	The file to which changes are being propagated (also called the *target* file). This file in the client workspace is overwritten by the result when you resolve.
`theirs`	The file from which changes are read (also known as the *source* file). This file resides in the depot, and is not changed by the resolve process.
`base`	The last integrated revision of the source file. When you use `integrate` to create the branched copy of the file in the depot, the newly-branched copy is *`base`*.

The integration algorithm

`p4 integrate` performs the following steps:

1. Apply the branch view to any target files provided on the command line to produce a list of source/target file pairs. If no files are provided on the command line, a list of all source/target file pairs is generated, including each revision of each source file that is to be integrated.

2. Discard any source/target pairs for which the source file revisions have already been integrated. Each revision of each file that has been integrated is recorded, to avoid making you merge changes more than once.
3. Discard any source/target pairs whose source file revisions have integrations pending in files that are already opened in the client.
4. Integrate all remaining source/target pairs. The target file is opened on the client for the appropriate action and merging is scheduled.

Integrate's actions

The integrate command will take one of three actions, depending on particular characteristics of the source and target files:

Action	Meaning
`branch`	If the target file does not exist, it is opened for `branch`. The branch action is a variant of add, but Perforce keeps a record of which source file the target file was branched from. This allows three-way merges to be performed between subsequent source and target revisions with the original source file revision as *`base`*.
`integrate`	If both the source and target files exist, the target is opened for integration, which is a variant of *`edit`*. Before a user can submit a file that has been opened for integration, the source and target must be merged with `p4 resolve`.
`delete`	When the target file exists but no corresponding source file is mapped through the branch view, the target is marked for deletion. This is consistent with `integrate`'s semantics: it attempts to make the target tree reflect the source tree.

By default, when you integrate using a branch specification, the original codeline contains the source files, and the branched codeline is the target. However, if you reverse the direction of integration by specifying the `-r` flag, the branched codeline contains the source, and the original files are the targets.

Integration Reporting

The branching-related reporting commands are:

Command	Function
`p4 integrate -n [filepatterns]`	Previews the results of the specified integration, but does not perform the integration. (To perform the integration, omit the `-n` flag.)
`p4 resolve -n [filepatterns]`	Displays files that are scheduled for resolve by `p4 integrate`, but does not perform the resolve. (To perform the resolve, omit the `-n` flag.)
`p4 resolved`	Displays files that have been resolved but not yet submitted.
`p4 branches`	Displays all branches.
`p4 integrated filepatterns`	Displays the integration history of the specified files.
`p4 filelog -i [filepatterns]`	Displays the revision histories of the specified files, including the integration histories of files from which the specified files were branched.

For More Information

Although Perforce's branching mechanism is relatively simple, the theory of branching can be very complex. When should a branch be created? At what point should code changes be propagated from one codeline to another? Who is responsible for performing merges? These questions will arise no matter what SCM system you're using, and the answers are not simple. Three on-line documents can provide some guidance in these matters.

A white paper on *InterFile Branching*, which describes Perforce's branching mechanism in technical detail, is available from:

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

Christopher Seiwald and Laura Wingerd's Best SCM Practices paper provides a discussion of many source configuration management issues, including an overview of basic branching techniques. This paper is available at:

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

Streamed Lines: Branching Patterns for Parallel Software Development is an extremely detailed paper on branching techniques. You'll find it at:

`http://www.enteract.com/~bradapp/acme/plop98/streamed-lines.html`

Chapter 10 Job Tracking

A *job* is a written description of some modification to be made to a source code set. A job might be a bug description, like "the system crashes when I press `return`", or it might be a system improvement request, like "please make the program run faster."

Whereas a job represents work that is intended, a changelist represents work actually done. Perforce's job tracking mechanism allows jobs to be linked to the changelists that implement the work requested by the job. A job can later be looked up to determine if and when it was fixed, which file revisions implemented the fix, and who fixed it. A job linked to a particular changelist is marked as completed when the changelist is submitted.

Jobs perform no functions internally to Perforce; rather, they are provided as a method of keeping track of information such as what changes to the source are needed, which user is responsible for implementing the job, and which file revisions contain the implementation of the job. The type of information tracked by the jobs system can be customized; fields in the job form can be added, changed, and deleted by Perforce administrators.

Job Usage Overview

There are five related but distinct aspects of using jobs.

- The Perforce superuser or administrator decides what fields are to be tracked in your system's jobs, the possible values of a job's fields, and their default values. This job template is edited with the `p4 jobspec` command. (See the *Perforce System Administrator's Guide* for details on how to edit the job specification. The job specification need not be changed before users can create jobs).
- Jobs can be created and edited by any user with `p4 job`.
- The `p4 jobs` command can be used to look up all the jobs that meet specified criteria.
- Jobs can be linked to changelists automatically or manually; when a job is linked to a changelist, the job is marked as `closed` when the changelist is submitted.
- The jobs that have been fixed can be displayed with Perforce reporting commands. These commands can list all jobs that fixed particular files or file revisions, all the jobs that were fixed in a particular changelist, or all the changelists that were linked to a particular job fix.

The remainder of this chapter discusses how these tasks are accomplished.

Creating and editing jobs using the default job specification

Jobs are created with the `p4 job` command.

Example: *Creating a job*

Sarah's Perforce server uses Perforce's default jobs specification. Sarah knows about a job in Elm's filtering subsystem, and she knows that Ed is responsible for Elm filters. Sarah creates a new job with `p4 job` *and fills in the resulting form as follows:*

```
Job:    new
User:   edk
Status: new
Date:   1998/05/18 17:15:40
Description:
        Filters on the "Reply-To:" field
        don't work.
```

Sarah has filled in a description and has changed `User:` *to* `edk`.

Since job fields differ from site to site, the fields in jobs at your site may be very different than what you see above. The default `p4 job` form's fields are:

Field Name	Description	Default
`Job`	The name of the job. Whitespace is not allowed in the name.	`new`
`User`	The user whom the job is assigned to, usually the username of the person assigned to fix this particular problem.	Perforce username of the person creating the job.
`Status`	`open`, `closed`, `suspended`, or `new`. An `open` job is one that has been created but has not yet been fixed. A `closed` job is one that has been completed. A `suspended` job is an open job that is not currently being worked on. Jobs with status `new` exist only while a new job is being created; they change to status `open` as soon as the form has been completed and the job added to the database.	`new`; changes to `open` after job creation form is closed.

Field Name	Description	Default
`Date`	The date the job was created, displayed as *`YYYY/MM/DD HH/MM/SS`*	The date and time at the moment this job was created. Changes to this field are ignored.
`Description`	Arbitrary text assigned by the user. Usually a written description of the problem that is meant to be fixed.	Text that *must* be changed

If `p4 job` is called with no parameters, a new job is created. The name that appears on the form is `new`, but this can be changed by the user to any desired string. If the `Job:` field is left as `new`, Perforce will assign the job the name `job`*`N`*, where *`N`* is a sequentially-assigned six-digit number.

Existing jobs can be edited with `p4 job` *`jobname`*. The user and description can be changed arbitrarily; the status can be changed to any of the three valid status values `open`, `closed`, or `suspended`. When you call `p4 job` *`jobname`* with a nonexistent *`jobname`*, Perforce creates a `new` job. (A job, if submitted with a `Status:` of `new`, has this status automatically changed to `open` upon completion of the job form.)

Creating and editing jobs with custom job specifications

A Perforce administrator can add and change fields within your server's jobs template with the `p4 jobspec` command. If this has been done, there may be additional fields in your `p4 jobs` form, and the names of the fields described above may have changed.

A sample customized job specification might look like this:

```
# Sample customized jobspec
#
# Job:           Job number
# Type:          The type of request: "bug" or "feature"
# Status:        Has it been fixed: "open", "closed", or "inprogress"
# Priority:      How soon should this job be fixed?
#                Values are "a", "b", "c", or "unknown"
# Owned_by:      Who's fixing the bug
# Reported_by:   Who reported the bug
# Reported_date: When the bug was first entered
# Mod_date:      Last time the bug was updated
# Description:   Textual description of the bug

Job:    new
Type:   unknown
Status: open
Priority:       unknown
User:           unowned
Reported_by:    edk
Reported_date:  2001/06/02 10:09:46
Mod_date:       2001/06/03 14:22:38
Description:
        <enter description here>
```

Some of the fields have been set by the administrator to allow one of a set of values; for example, `Priority:` must be one of `a`, `b`, `c`, or `unknown`. The `p4 job` fields don't tell you what the valid values of the fields are; your Perforce administrator can tell you this in comments at the top of the job form. If you find the information in the comments for your jobs to be insufficient to enter jobs properly, please tell your Perforce administrator.

Viewing jobs by content with jobviews

Jobs can be reported with `p4 jobs`. In its simplest form, with no arguments, `p4 jobs` will list every job stored in your Perforce server. However, `p4 job -e` *`jobview`* will list all jobs that match the criteria contained in *`jobview`*.

Throughout the following discussion, the following rules apply:

- Textual comparisons within jobviews are case-insensitive, as are the field names that appear in jobviews,
- only alphanumeric text and punctuation can appear in a jobview,
- there is currently no way to search for particular phrases. Jobviews can search jobs only by individual words.

Finding jobs containing particular words

The jobview `'word1 word2 ... wordN'` can be used to find jobs that contain all of *word1* through *wordN* in any field (excluding date fields).

Example: *Finding jobs that contain all of a set of words in any field.*

Ed wants to find all jobs that contain the words `filter`, *`file`, and* `mailbox`. *He types:*

```
p4 jobs -e 'filter file mailbox'
```

Spaces between search terms in jobviews act as boolean *and*'s. You can use ampersands instead of spaces in jobviews, so the jobviews `'joe sue'` and `'joe&sue'` are identical.

To find jobs that contain any of the terms, separate the terms with the '|' character.

Example: *Finding jobs that contain any of a set of words in any field.*

Ed wants to find jobs that contains any of the words `filter`, `file` *or* `mailbox`. *He types:*

```
p4 jobs -e 'filter|file|mailbox'
```

Finding jobs by field values

Search results can be narrowed by matching values within specific fields with the jobview syntax `'fieldname=value'`. Value must be a single alphanumeric word.

Example: *Finding jobs that contain words in specific fields*

Ed wants to find all open jobs related to filtering of which he is the owner. He types:

```
p4 jobs -e 'status=open user=edk filter.c'
```

This will find all jobs with a `Status:` *of* `open`, *a* `User:` *of* `edk`, *and the word* `filter.c` *in any non-date field.*

Using and escaping wildcards in jobviews

The wildcard "`*`" allows for partial word matches. The jobview "*`fieldname`*`=string*`" matches "`string`", "`stringy`", "`stringlike`", and so on.

To search for words that happen to contain wildcards, escape them at the command line. For instance, to search for "`*string`" (perhaps in reference to `char *string`), you'd use the following:

```
p4 jobs -e '\*string'
```

Negating the sense of a query

The sense of a search term can be reversed by prefacing it with `^`, the *not* operator.

Example: *Finding jobs that don't contain particular words.*

Ed wants to find all open jobs related to filtering of which he is not the owner. He types:

```
p4 jobs -e 'status=open ^user=edk filter'
```

This displays all jobs with a `Status:` *of* `open`*, a* `User:` *of anyone but* `edk`*, and the word* `filter` *in any non-date field.*

The *not* operator `^` can be used only directly after an *and* (`space` or `&`). Thus, `p4 jobs -e '^user=edk'` is not allowed.

You can use the `*` wildcard to get around this: `p4 jobs -e 'job=* ^user=edk'` returns all jobs with a `user` field not matching `edk`.

Using dates in jobviews

Jobs can be matched by date by expressing the date as *`yyyy/mm/dd`* or *`yyyy/mm/dd:hh:mm:ss`*. If you don't provide a specific time, the equality operator `=` matches the entire day.

Example: *Using dates within jobviews.*

Ed wants to view all jobs modified on July 13, 1998. He enters

```
p4 jobs -e 'mod_date=1998/07/13'
```

Comparison operators and field types

The usual comparison operators are available. They are:

```
=          >          <          >=         <=
```

The behavior of these operators depends upon the type of the field in the jobview.The field types are:

Field Type	Explanation	Examples
`word`	A single word	A user name: `edk`
`text`	A block of text	A job's description
`line`	A single line of text. Differs from text fields only in that line values are entered on the same line as the field name, and text values are entered on the lines beneath the field name.	An email address A user's real name: `Linda Hopper`

Field Type	Explanation	Examples
`select`	One of a set of values	A job's status: `open/suspended/closed`
`date`	A date value	The date and time of job creation: `1998/07/15:13:21:4`

Field types are often obvious from context; a field called *`mod_date`*, for example, is most likely a date field. If you're not sure of a field's type, run `p4 jobspec -o`, which outputs the job specification your local jobspec. The field called `Fields:` lists the job fields' names and datatypes.

The jobview comparison operators behave differently depending upon the type of field they're used with. The comparison operators match the different field types as follows:

Field Type	Use of Comparison Operators in Jobviews
`word`	The equality operator `=` must match the value in the word field exactly. The inequality operators perform comparisons in ASCII order.
`text`	The equality operator `=` matches the job if the word given as the value is found anywhere in the specified field. The inequality operators are of limited use here, since they'll match the job if *any* word in the specified field matches the provided value. For example, if a job has a text field `ShortDescription:` that contains only the phrase `gui bug`, and the jobview is `'ShortDesc<filter'`, the job will match the jobview, because `bug<filter`.
`line`	See `text`, above.
`select`	The equality operator `=` matches a job if the value of the named field is the specified word. Inequality operators perform comparisons in ASCII order.
`date`	Dates are matched chronologically. If a specific time is not provided, the operators `=`, `<=`, and `>=` will match the whole day.

Linking Jobs to Changelists

Perforce automatically changes the value of a job's status field to `closed` when the job is linked to a particular changelist, and the changelist is submitted.

Jobs can be linked to changelists in one of two ways:

- Automatically, by setting the `JobView:` field in the `p4 user` form to a jobview that matches the job, and
- manually, with the `p4 fix` command.

Automatically linking jobs to changelists with the p4 user form

The `p4 user` form can be used to automatically include particular jobs on any new changelists created by that user. To do this, call `p4 user` and change the `JobView:` field value to any valid jobview.

Example: *Automatically linking jobs to changelists with the p4 user form's JobView field.*

Ed wants to see all open jobs that he owns in all changelists he creates. He types `p4 user` *and adds a* `JobView:` *field:*

```
User:      edk
Update:    1998/06/02 13:11:57
Access:    1998/06/03 20:11:07
JobView:   user=edk&status=open
```

All of Ed's jobs that meet these `JobView:` *criteria will automatically appear on all changelists he creates. He can, and should, delete jobs that aren't fixed by the changelist from the changelist form before submission. When a changelist is submitted, the jobs linked to it will have their* `status:` *field's value changed to* `closed`.

Automatic update of job status

The value of a job's `status` field is automatically changed to `closed` when one of its associated changelists is successfully submitted. Jobs can be disassociated from changelists by deleting the job from the changelist's change form. Similarly, any job can be added to a changelist by adding it to a changelist's change form.

Example: *Including and excluding jobs from changelists.*

Ed has set his `p4 user`*'s* `JobView:` *field as in the example above. He is unaware of a job that Sarah has made Ed the owner of (when she entered the job, she set the* `User:` *field to* `edk`*). He is currently working on an unrelated problem; he types* `p4 submit` *and sees the following:*

```
Change: new
Client: edk
User:   edk
Status: new
Description:
        Updating "File" files
Jobs:
        job000125        # Filters on "Reply-To" field don't work

Files:
        //depot/src/file.c         # edit
        //depot/src/file_util.c    # edit
        //depot/src/fileio.c       # edit
```

Since this job is unrelated to the work he's been doing, and since it hasn't been fixed, he deletes it from the form and then quits from the editor. The changelist is submitted, and the job is not associated with it.

Example: *Submit a changelist with an attached job.*

Ed uses the reporting commands to read the details about the job. He fixes this problem, and a number of other filtering bugs; when he next types `p4 submit`*, he sees:*

```
Change: new
Client: edk
User:   rlo
Status: new
Description:
        Fixes a number of filter problems
Jobs:
        job000125          # Filters on "Reply-To" field don't work
Files:
        //depot/filter/actions.c    # edit
        //depot/filter/audit.c      # edit
        //depot/filter/filter.c     # edit
```

Since the job is fixed in this changelist, Ed leaves the job on the form. When he quits from the editor, the job's status is changed to `closed`*.*

Manually associating jobs with changelists

`p4 fix -c` *`changenum jobname`* can be used to link any job to any changelist. If the changelist has already been submitted, the value of the job's `Status:` field is changed to `closed`. Otherwise, the job keeps its current status.

Example: *Manually attaching jobs to changelists.*

Sarah has submitted a job called `options-bug` *to Ed. Unbeknownst to Sarah, the bug reported by the job was fixed in Ed's previously submitted changelist 18. Ed links the job to the previously submitted changelist with* `p4 fix -c 18 options-bug`. *Since the changelist has already been submitted, the job's status is changed to* `closed`.

It is never necessary to use `p4 fix` to link an open job to a changelist newly created by the owner of the job, since this is done automatically. However, you can use `p4 fix` to link a changelist to a job owned by another user.

What if there's no status field?

The discussion in this section has assumed that the server's job specification still contains the default `Status:` field. If the job specification has been altered so that this is no longer true, jobs can still be linked to changelists, but nothing in the job changes when the changelist is submitted. (In most cases, this is not a desired form of operation.) Please see the chapter on editing job specifications in the *Perforce System Administrator's Guide* for more details.

Deleting Jobs

A job can be unlinked from any changelist with `p4 fix -d -c` *`changenum jobname`*.

Jobs can be deleted entirely with `p4 job -d` *`jobname`*.

Integrating with External Defect Tracking Systems

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

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

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

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

Even if you don't use the P4DTI kit as a starting point, you can still use Perforce's job system as the interface between Perforce and your defect tracker. See the *Perforce System Administrator's Guide* for more information.

Job Reporting Commands

The job reporting commands can be used to show the correspondence between files and the jobs they fix.

To See a Listing of...	Use This Command:
...all jobs that match particular criteria	`p4 jobs -e jobview`
...all the jobs that were fixed by changelists that affected particular file(s)	`p4 jobs filespec`
...all changelists and file revisions that fixed a particular job	`p4 fixes -j jobname`
...all jobs linked to a particular changelist	`p4 fixes -c changenum`
...all jobs fixed by changelists that contain particular files or file revisions	`p4 fixes filespec`

Other job reporting variations are available. For more examples, please see “Job Reporting” on page 131, or consult the *Perforce Command Reference*.

Chapter 11 Reporting and Data Mining

Perforce's reporting commands supply information on all data stored within the depot. There are many reporting commands; in fact, there are almost as many reporting commands as there are action commands. These commands have been discussed throughout this manual; this chapter presents the same commands and provides additional information for each command. Tables in each section contain answers to questions of the form "*How do I find information about...?*"

Many of the reporting commands have numerous options, but discussion of all options for each command is beyond the scope of this manual. For a full description of any particular command, please consult the *Perforce Command Reference*, or type `p4 help` *`command`* at the command line.

One previously mentioned note on syntax is worth repeating here: any *`filespec`* argument in Perforce commands, as in:

```
p4 files filespec
```

will match any file pattern that is supplied in local syntax, depot syntax, or client syntax, with any Perforce wildcards. Brackets around `[filespec]` mean that the file specification is optional. Additionally, many of the reporting commands can take revision specifiers as part of the filespec. Revision specifiers are discussed on "Specifying Older File Revisions" on page 51.

Files

The commands that report on files fall into two categories: those that give information about file *contents*, (for instance, `p4 print`, `p4 diff`), and those that supply information on file *metadata*, the data that describe a file with no reference to content (e.g. `p4 files`, `p4 filelog`). The first set of reporting commands discussed in this section describe file metadata, while the second set describe file contents.

File metadata

Basic file information

To view information about single revisions of one or more files, use `p4 files`. This command provides the locations of the files within the depot, the actions (`add`, `edit`, `delete`, and so on) on those files at the specified revisions, the changelists the specified file revisions were submitted in, and the files' types. The output has this appearance:

```
//depot/README#5 - edit change 6 (text)
```

The `p4 files` command requires one or more *`filespec`* arguments. Filespec arguments can, as always, be provided in Perforce or local syntax, but the output always reports on the corresponding files within the depot. If you don't provide a revision number, Perforce uses the head revision.

Unlike most other commands, `p4 files` also describes deleted revisions, rather than suppressing information about deleted files.

To View File Metadata for...	Use This Command:
...all files in the depot, whether or not visible through your client view	`p4 files //depot/...`
...all the files currently in any client workspace	`p4 files @clientname`
...all the files in the depot that are mapped through your current client workspace view	`p4 files //clientname/...`
...a particular set of files in the current working directory	`p4 files filespec`
...a particular file at a particular revision number	`p4 files filespec#revisonNum`
...all files at change *n*, whether or not the file was actually included in change *n*	`p4 files @n`
...a particular file within a particular label	`p4 files filespec@labelname`

File revision history

The revision history of a file is provided by `p4 filelog`. One or more file arguments must be provided, and since the point of `p4 filelog` is to list information about each revision of particular files, file arguments to this command may not contain a revision specification.

The output of `p4 filelog` has this form:

```
... #3 change 23 edit on 1997/09/26 by edk@doc <ktext> 'Fix help system'
... #2 change 9 edit on 1997/09/24 by lisag@src <text> 'Change file'
... #1 change 3 add on 1997/09/24 by edk@doc <text> 'Added filtering bug'
```

For each file that matches the *`filespec`* argument, the complete list of file revisions is presented, along with the number of the changelist that the revision was submitted in, the date of submission, the user who submitted the revision, the file's type at that revision, and the first few characters of the changelist description. With the `-l` flag, the entire description of each changelist is printed:

```
#3 change 23 edit on 1997/09/26 by edk@doc
Updated help files to reflect changes
in filtering system & other subsystems
...<etc.>
```

Opened files

To see which files are currently opened within a client workspace, use `p4 opened`. For each opened file within the client workspace that matches a file pattern argument, `p4 opened` prints a line like the following:

```
//depot/elm_proj/README - edit default change (text)
```

Each opened file is described by its depot name and location, the operation that the file is opened for (`add`, `edit`, `delete`, `branch`, or `integrate`), which changelist the file is included in, and the file's type.

To See...	Use This Command:
...a listing of all opened files in the current workspace	`p4 opened`
...a list of all opened files in all client workspaces	`p4 opened -a`
...a list of all files in a numbered pending changelist	`p4 opened -c` *`changelist#`*
...a list of all files in the default changelist	`p4 opened -c default`
...whether or not a specific file is opened by you	`p4 opened` *`filespec`*
...whether or not a specific file is opened by anyone	`p4 opened -a` *`filespec`*

Relationships between client and depot files

It is often useful to know how the client and depot are related at a particular moment in time. Perhaps you simply want to know where a particular client file is mapped to within the depot, or you may want to know whether or not the head revision of a particular depot file has been copied to the client. The commands that express the relationship between client and depot files are `p4 where`, `p4 have`, and `p4 sync -n`. The first of these commands, `p4 where`, shows the mappings between client workspace files, depot files, and local OS syntax. `p4 have` tells you which revisions of files you've last synced to your client workspace, and `p4 sync -n` describes which files would be read into your client workspace the next time you perform a `p4 sync`.

All these commands can be used with or without *`filespec`* arguments. `p4 sync -n` is the only command in this set that allows revision specifications on the *`filespec`* arguments.

The output of `p4 where` *`filename`* looks like this:

```
//depot/elm_proj/doc/Ref.guide //edk/doc/Ref.guide /usr/edk/doc/Ref.guide
```

The first part of the output is the location of the file in depot syntax; the second part is the location of the same file in client syntax, and the third is the location of the file in local OS syntax.

`p4 have`'s output has this form:

```
//depot/doc/Ref.txt#3 - /usr/edk/elm/doc/Ref.txt
```

and `p4 sync -n` provides output like:

```
//depot/doc/Ref.txt#3 - updating /usr/edk/elm/doc/Ref.txt
```

The following table lists other useful commands:

To See...	Use This Command:
...which revisions of which files you have in the client workspace	`p4 have`
...which revision of a particular file is in your client workspace	`p4 have` *`filespec`*
...where a particular file maps to within the depot, the client workspace, and the local OS	`p4 where` *`filespec`*
...where a particular file in the depot maps to in the workspace	`p4 where //depot/.../`*`filespec`*
...which files would be synced into your client workspace from the depot when you do the next sync	`p4 sync -n`

File contents

Contents of a single revision

You can view the contents of any file revision within the depot with `p4 print`. This command simply prints the contents of the file to standard output, or to the specified output file, along with a one-line banner that describes the file. The banner can be removed by passing the `-q` flag to `p4 print`. When printed, the banner has this format:

```
//depot/elm_proj/README#23 - edit change 50 (text)
```

`p4 print` takes a mandatory file argument, which can include a revision specification. If a revision is specified, the file is printed at the specified revision; if no revision is specified, the head revision is printed.

To See the Contents of Files...	Use This Command:
...at the current head revision	`p4 print` *`filespec`*
...without the one-line file header	`p4 print -q` *`filespec`*
...at a particular change number	`p4 print` *`filespec@changenum`*

Annotated file contents

Use `p4 annotate` to find out which file revisions changed which lines in a file.

By default, `p4 annotate` displays the file, each line of which is prepended by a revision number indicating the revision that made the change. The `-a` option allows you to display all lines, including lines no longer present at the head revision.

Example: *Using* `p4 annotate` *to track changes to a file*

A file is added (`file.txt#1`*) to the depot, containing the following lines:*

```
This is a text file.
The second line has not been changed.
The third line has not been changed.
```

The file is edited so that `file.txt#2` *reads:*

```
This is a text file.
The second line is new.
```

A third changelist is submitted, that includes no changes to `file.txt`*. After the third changelist, the output of* `p4 annotate` *looks like this:*

```
$ p4 annotate file.txt
//depot/files/file.txt#3 - edit change 12345 (text)
1: This is a text file.
2: The second line is new.
```

The first line of file.txt has been present since `file.txt#1`*, and the next two lines have been present since revision* `#2`*.*

To show all lines (including deleted lines) in the file, use `p4 annotate -a` *as follows:*

```
$ p4 annotate -a file.txt
//depot/files/file.txt#3 - edit change 12345 (text)
1-3: This is a text file.
1-1: The second line has not been changed.
1-1: The third line has not been changed.
2-3: The second line is new.
```

The contents of `file.txt` *are displayed in chunks. The first chunk consists of the first line, which was present for revisions* `#1` *through* `#3` *(*`1-3`*). The second chunk consists of the two lines that existed only at revision* `#1` *(*`1-1`*). The third chunk shows the line that replaced those in the second chunk, and that it was present in the file for revisions* `#2` *through* `#3` *(*`2-3`*).*

File content comparisons

A client workspace file can be compared to any revision of the same file within in the depot with `p4 diff`. This command takes a filespec argument; if no revision specification is supplied, the workspace file is compared against the revision last read into the workspace.

The `p4 diff` command has many options available; only a few are described in the table below. For more details, please consult the *Perforce Command Reference*.

Whereas `p4 diff` compares a client workspace file against depot file revisions, `p4 diff2` can be used to compare *any* two revisions of a file. It can even be used to compare revisions of different files. `p4 diff2` takes two file arguments -- wildcards are allowed, but any wildcards in the first file argument must be matched with a corresponding wildcard in the second. This makes it possible to compare entire trees of files.

There are many more flags to `p4 diff` than described below. For a full listing, please type `p4 help diff` at the command line, or consult the *Perforce Command Reference*.

To See the Differences between...	Use This Command:
...an open file within the client workspace and the revision last taken into the workspace	`p4 diff file`
...any file within the client workspace and the revision last taken into the workspace	`p4 diff -f file`
...a file within the client workspace and the same file's current head revision	`p4 diff file#head`
...a file within the client workspace and a specific revision of the same file within the depot	`p4 diff file#revnumber`
...the *n*-th and head revisions of a particular file	`p4 diff2 filespec filespec#n`
...all files at changelist *n* and the same files at changelist *m*	`p4 diff2 filespec@n filespec@m`

To See the Differences between...	Use This Command:
...all files within two branched codelines	`p4 diff2 //depot/path1/... //depot/path2/...`
...a file within the client workspace and the revision last taken into the workspace, passing the context diff flag to the underlying diff	`p4 diff -dc file`

The last example above bears further explanation; to understand how this works, it is necessary to discuss how Perforce implements and calls underlying diff routines.

Perforce uses two separate diffs: one is built into the `p4d` server, and the other is used by the `p4` client. Both diffs contain identical, proprietary code, but are used by separate sets of commands. The client diff is used by `p4 diff` and `p4 resolve`, and the server diff is used by `p4 describe`, `p4 diff2`, and `p4 submit`.

Perforce's built-in diff routine allows three `-d<flag>` flags: `-du`, `-dc`, and `-dn`. Both `p4 diff` and `p4 diff2` allow any of these flags to be specified. These flags behave identically to the corresponding flags in the standard UNIX diff.

Although the server must always use Perforce's internal diff routine, the client diff can be set to any external diff program by pointing the `P4DIFF` environment variable to the full path name of the desired executable. Any flags used by the external diff can be passed to it with `p4 diff`'s `-d` flag. Flags are passed to the underlying diff according to the following rules:

- If the character immediately following the `-d` is not a single quote, then all the characters between the `-d` and whitespace are prepended with a dash and sent to the underlying diff.
- If the character immediately following the `-d` is a single quote, then all the characters between the opening quote and the closing quote are prepended with a dash and sent to the underlying diff.

The following examples demonstrate the use of these rules in practice.

If you want to pass the following flag to an external client diff program:	Then call `p4 diff` this way:
`-u`	`p4 diff -du`
`--brief`	`p4 diff -d-brief`
`-C 25`	`p4 diff -d'C 25'`

Changelists

Two separate commands are used to describe changelists. The first, `p4 changes`, lists changelists that meet particular criteria, without describing the files or jobs that make up the changelist. The second command, `p4 describe`, lists the files and jobs affected by a single changelist. These commands are described below.

Viewing changelists that meet particular criteria

To view a list of changelists that meet certain criteria, such as changelists with a certain status, or changelists that affect a particular file, use `p4 changes`.

The output looks like this:

```
Change 36 on 1997/09/29 by edk@eds_elm 'Changed filtering me'
Change 35 on 1997/09/29 by edk@eds_elm 'Misc bug fixes: fixe'
Change 34 on 1997/09/29 by lisag@lisa 'Added new header inf'
```

By default, `p4 changes` displays an aggregate report containing one line for every changelist known to the system, but command line flags and arguments can be used to limit the changelists displayed to those of a particular status, those affecting a particular file, or the last *n* changelists.

Currently, the output can't be restricted to changelists submitted by particular users, although you can write simple shell or Perl scripts to implement this (you'll find an example of such a script in the *Perforce System Administrator's Guide*).

To See a List of Changelists...	Use This Command:
...with the first 31 characters of the changelist descriptions	`p4 changes`
...with the complete description of each changelist	`p4 changes -l`
...including only the last *n* changelists	`p4 changes -m n`
...with a particular status (`pending` or `submitted`)	`p4 changes -s status`
...from a particular user	`p4 changes -u user`
...from a particular client workspace	`p4 changes -c workspace`
...limited to those that affect particular files	`p4 changes filespec`
...limited to those that affect particular files, but including changelists that affect files which were later integrated with the named files	`p4 changes -i filespec`

To See a List of Changelists...	Use This Command:
...limited to changelists that affect particular files, including only those changelists between revisions *m* and *n* of these files	`p4 changes` *filespec*`#`*m*`,#`*n*
...limited to those that affect particular files at each files revisions between labels *lab1* and *lab2*	`p4 changes` *filespec*`@`*lab1*`,@`*lab2*
...limited to those between two dates	`p4 changes @`*date1*`,@`*date2*
...between an arbitrary date and the present day	`p4 changes @`*date1*`,@now`

Note For details about Perforce commands that allow you to use revision ranges with file specifications, see "Revision Ranges" on page 54.

Files and jobs affected by changelists

To view a list of files and jobs affected by a particular changelist, along with the *diffs* of the new file revisions and the previous revisions, use `p4 describe`.

The output of `p4 describe` looks like this:

```
Change 43 by lisag@warhols on 1997/08/29 13:41:07

        Made grammatical changes to basic Elm documentation

Jobs fixed...

job000001 fixed on 1997/09/29 by edk@edk
        Fix grammar in main Elm help file

Affected files...

... //depot/doc/elm.1#2 edit

Differences...

==== //depot/doc/elm.1#2 (text) ====
53c53
> Way number 2, what is used common-like when, you know, like
---
> The second method is commonly used when transmitting

...<etc.>
```

This output is quite lengthy, but a shortened form that eliminates the diffs can be generated with `p4 describe -s` *changenum*.

To See:	Use This Command:
...a list of files contained in a pending changelist	`p4 opened -c` *changelist#*
...a list of all files submitted and jobs fixed by a particular changelist, displaying the diffs between the file revisions submitted in that changelist and the previous revisions	`p4 describe` *changenum*
...a list of all files submitted and jobs fixed by a particular changelist, without the file diffs	`p4 describe -s` *changenum*
...a list of all files and jobs affected by a particular changelist, while passing the context diff flag to the underlying diff program	`p4 describe -dc` *changenum*
...the state of particular files at a particular changelist, whether or not these files were affected by the changelist	`p4 files` *filespec@changenum*

For more commands that report on jobs, see "Job Reporting" on page 131.

Labels

Reporting on labels is accomplished with a very small set of commands. The only command that reports only on labels, `p4 labels`, prints its output in the following format:

```
Label release1.3 1997/5/18 'Created by edk'
Label lisas_temp 1997/10/03 'Created by lisag'
...<etc.>
```

The other label reporting commands are variations of commands we've seen earlier.

To See:	Use This Command:
...a list of all labels, the dates they were created, and the name of the user who created them	`p4 labels`
...a list of all labels containing a specific revision (or range)	`p4 labels` *file#revrange*
...a list of files that have been included in a particular label with `p4 labelsync`	`p4 files @`*labelname*
...what `p4 sync` would do when retrieving files from a particular label into your client workspace	`p4 sync -n @`*labelname*

Branch and Integration Reporting

The plural form command of branch, `p4 branches`, lists the different branches in the system, along with their owners, dates created, and descriptions. Separate commands are used to list files within a branched codeline, to describe which files have been integrated, and to perform other branch-related reporting.

The table below describes the most commonly used commands for branch- and integration-related reporting.

To See:	Use This Command:
...a list of all branches known to the system	`p4 branches`
...a list of all files in a particular branched codeline	`p4 files` *`filespec`*
...what a particular `p4 integrate` variation would do, without actually doing the integrate	`p4 integrate [`*`args`*`] -n [`*`filespec`*`]`
...a list of all the revisions of a particular file	`p4 filelog -i` *`filespec`*
...what a particular `p4 resolve` variation would do, without actually doing the resolve	`p4 resolve [`*`args`*`] -n [`*`filespec`*`]`
...a list of files that have been resolved but have not yet been submitted	`p4 resolved [`*`filespec`*`]`
...a list of integrated, submitted files that match the *`filespec`* arguments	`p4 integrated` *`filespec`*
...a list of all the revisions of a particular file, including revision of the file(s) it was branched from	`p4 filelog -i` *`filespec`*

Job Reporting

Two commands report on jobs. The first, `p4 jobs`, reports on all jobs known to the system, while the second command, `p4 fixes`, reports only on those jobs that have been attached to changelists. Both of these commands have numerous options.

Basic job information

To see a list of all jobs known to the system, use `p4 jobs`. This command produces output similar to the following:

```
job000302 on 1997/08/13 by saram *open* 'FROM: headers no'
filter_bug on 1997/08/23 by edk *closed* 'Can't read filters w'
```

Its output includes the job's name, date entered, job owner, status, and the first 31 characters of the job description.

All jobs known to the system are displayed unless command-line options are supplied. These options are described in the table below.

To See a List of Jobs:	Use This Command:
...including all jobs known to the server	`p4 jobs`
...including the full texts of the job descriptions	`p4 jobs -l`
...for which certain fields contain particular values (For more about jobviews, see "Viewing jobs by content with jobviews" on page 112)	`p4 jobs -e jobview`
...that have been fixed by changelists that contain specific files	`p4 jobs filespec`
...that have been fixed by changelists that contain specific files, including changelists that contain files that were later integrated into the specified files	`p4 jobs -i filespec`

Jobs, fixes, and changelists

Any jobs that have been linked to a changelist with `p4 change`, `p4 submit`, or `p4 fix` is said to be *fixed*, and can be reported with `p4 fixes`.

The output of `p4 fixes` looks like this:

```
job000302 fixed by change 634 on 1997/09/01 by edk@eds_elm
filter_bug fixed by change 540 on 1997/10/22 by edk@eds_elm
```

A number of options allow the reporting of only those changes that fix a particular job, jobs fixed by a particular changelist, or jobs fixed by changelists that are linked to particular files.

A fixed job will not necessarily have a status of `closed` job, since `open` jobs can be linked to pending changelists, and `pending` jobs can be reopened even after the associated changelist has been submitted. To list jobs with a particular status, use `p4 jobs`.

To See a Listing of...	Use This Command:
...all fixes for all jobs	`p4 fixes`
...all changelists linked to a particular job	`p4 fixes -j jobname`
...all jobs linked to a particular changelist	`p4 fixes -c changenum`

To See a Listing of...	Use This Command:
...all jobs fixed by changelists that contain particular files	`p4 fixes` *filespec*
...all jobs fixed by changelists that contain particular files, including changelists that contain files that were later integrated with the specified files	`p4 fixes -i` *filespec*

Reporting for Daemons

The Perforce change review mechanism uses the following reporting commands. Any of these commands might also be used with user-created daemons. For further information on daemons, please see the *Perforce System Administrator's Guide.*

To list...	Use this Command:
...the names of all counter variables currently used by your Perforce system	`p4 counters`
...the numbers of all changelists that have not yet been reported by a particular counter variable	`p4 review -t` *countername*
...all users who have subscribed to review particular files	`p4 reviews` *filespec*
...all users who have subscribed to read any files in a particular changelist	`p4 reviews -c` *changenum*
...a particular user's email address	`p4 users` *username*

System Configuration

Three commands report on the Perforce system configuration. One command reports on all Perforce users, another prints data describing all client workspaces, and a third reports on Perforce depots.

`p4 users` generates its data as follows:

```
edk <edk@eds_ws> (Ed K.) accessed 1997/07/13
lisag <lisa@lisas_ws> (Lisa G.) accessed 1997/07/14
```

Each line includes a username, an email address, the user's "real" name, and the date that Perforce was last accessed by that user.

To report on client workspaces, use `p4 clients`:

```
Client eds_elm 1997/09/12 root /usr/edk 'Ed's Elm workspace'
Client lisa_doc 1997/09/13 root /usr/lisag 'Created by lisag.'
```

Each line includes the client name, the date the client was last updated, the client root, and the description of the client.

Depots can be reported with `p4 depots`. All depots known to the system are reported on; the described fields include the depot's name, its creation date, its type (`local` or `remote`), its IP address (if `remote`), the mapping to the local depot, and the system administrator's description of the depot.

The use of multiple depots on a single Perforce server is discussed in the *Perforce System Administrator's Guide.*

To view:	Use This Command:
...user information for all Perforce users	`p4 users`
...user information for only certain users	`p4 users` *username*
...brief descriptions of all client workspaces	`p4 clients`
...a list of all defined depots	`p4 depots`

Special Reporting Flags

Two special flags, `-o` and `-n`, can be used with certain action commands to change their behavior from action to reporting.

The `-o` flag is available with most of the Perforce commands that normally bring up forms for editing. This flag tells these commands to write the form information to standard output, instead of bringing the definition into the user's editor. This flag is supported by the following commands:

`p4 branch`	`p4 client`	`p4 label`
`p4 change`	`p4 job`	`p4 user`

The `-n` flag prevents commands from doing their job. Instead, the commands simply tell you what they would ordinarily do. You can use the `-n` flag with the following commands

`p4 integrate`	`p4 resolve`	`p4 labelsync`	`p4 sync`

Reporting with Scripting

Although Perforce's reporting commands are sufficient for most needs, there may be times when you want to view data in a format that Perforce doesn't directly support. In these situations, the reporting commands can be used in combination with scripts to print only the data that you want to see. Here are two examples.

Comparing the change content of two file sets

To compare the "change content" of two sets of files, you have to diff them externally. To do this, run `p4 changes` twice, once on each set of files, and then use any external diff routine to compare them.

In the following example, `main` represents the main codeline, and `r3.2` is a codeline that was originally branched from `main`:

```
p4 changes //depot/main/... > changes-in-main
p4 changes //depot/r3.2/... > changes-in-r98.4
diff changes-in-main changes-in-r98.4
```

You can use this to uncover which changes have been made to `r98.4` that haven't been integrated back into `main`.

Appendix A Installing Perforce

This appendix outlines how to install a Perforce server for the first time.

This appendix is mainly intended for people installing an evaluation copy of Perforce for trial use; if you're installing Perforce for production use, or are planning on extensive testing of your evaluation server, we strongly encourage you to read the detailed information in the *Perforce System Administrator's Guide.*

Getting Perforce

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

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

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

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

Installing Perforce on UNIX

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

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

Only a few additional steps need to be performed before `p4` and `p4d` can be run. They are described in detail in the following sections. Briefly:

- Download the files and make them executable,
- create a root directory for the Perforce files,
- provide a TCP/IP port to `p4d`,
- provide the hostname of the Perforce server and the `p4d` port number to the Perforce client program(s), and
- start the Perforce server (`p4d`).

Download the files and make them executable

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

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

Creating a Perforce server root directory

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

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

For security purposes, you are strongly advised not to run `p4d` as `root` or any other privileged user.

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

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

Telling the Perforce server which port to listen to

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

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

Starting the Perforce server

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

```
p4d &
```

Although this command is sufficient to run p4d, other flags (for instance, those that control such things as error logging, checkpointing, and journaling), can be provided. These flags (and others) are discussed in the *Perforce System Administrator's Guide*.

Stopping the Perforce server

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

```
p4 admin stop
```

to shut down the Perforce server.

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

With the introduction of p4 admin stop in Release 99.2, the practice of manually killing the p4d server has become obsolete.

Telling Perforce clients which port to talk to

By this time, your Perforce server should be up and running; see "Connecting to the Perforce Server" on page 21 for information on how to set up your environment to allow Perforce's client programs to talk to the server.

Installing Perforce on Windows

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

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

  This option allows for the installation of p4.exe (the Perforce Comnand-Line Client), p4win.exe (P4Win, the Perforce Windows Client), and p4scc.dll (Perforce's implementation of the Microsoft common SCM interface).

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

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

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

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

Terminology note: Windows services and servers

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

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

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

In most cases, you will want to install Perforce as a service, not a server. For a more detailed discussion of the distinction between the two options, see the *Perforce System Administrator's Guide.*

Starting and stopping Perforce on Windows

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

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

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

```
p4 admin stop
```

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

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

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

Appendix B Environment Variables

This table lists all the Perforce environment variables and their definitions.

You'll find a full description of each variable in the *Perforce Command Reference*.

Variable	Definition
P4CHARSET	For internationalized installations only, the character set to use for Unicode translations
P4CLIENT	Name of current client workspace
P4CONFIG	File name from which values for current environment variables are to be read
P4DIFF	The name and location of the diff program used by `p4 resolve` and `p4 diff`
P4EDITOR	The editor invoked by those Perforce commands that use forms
P4HOST	Name of host computer to impersonate. Only used if the `Host:` field of the current client workspace has been set in the `p4 client` form.
P4JOURNAL	A file that holds the database journal data, or `off` to disable journaling.
P4LANGUAGE	This variable is reserved for system integrators.
P4LOG	Name and path of the file to which Perforce server error and diagnostic messages are to be logged.
P4MERGE	A third-party merge program to be used by `p4 resolve`'s merge option
P4PAGER	The program used to page output from `p4 resolve`'s diff option
P4PASSWD	Stores the user's password as set in the `p4 user` form
P4PORT	For the Perforce server, the port number to listen on; for the `p4 client`, the name and port number of the Perforce server with which to communicate
PWD	The directory used to resolve relative filename arguments to `p4` commands
P4ROOT	Directory in which `p4d` stores its files and subdirectories
P4USER	The user's Perforce username
TMP	The directory to which Perforce writes its temporary files

Setting and viewing environment variables

Each operating system and shell has its own syntax for setting environment variables. The following table shows how to set the `P4CLIENT` environment variable in each OS and shell:

OS or Shell	Environment Variable Example
UNIX: `ksh`, `sh`, `bash`	`P4CLIENT=`*`value`* `; export P4CLIENT`
UNIX: `csh`	`setenv P4CLIENT` *`value`*
VMS	`def/j P4CLIENT "`*`value`*`"`
Mac MPW	`set -e P4CLIENT` *`value`*
Windows	`p4 set P4CLIENT=`*`value`* (See the `p4 set` section of the *Perforce Command Reference* or run the command `p4 help set` to obtain more information about setting Perforce's registry variables in Windows). Windows administrators running Perforce as a service can set variables for use by a specific service with `p4 set -S` *`svcname var=value`*.

To view a list of the values of all Perforce variables, use `p4 set` without any arguments.

On UNIX, this displays the values of the associated environment variables. On NT, this displays either the MS-DOS environment variable (if set), or the value in the registry and whether it was defined with `p4 set` (for the current user) or `p4 set -s` (for the local machine).

Appendix C Glossary

Term	Definition
access level	A permission assigned to a user to control which Perforce commands the user can execute. See *protections*.
admin access	An access level that gives the user permission to run Perforce commands that override *metadata*, but do not affect the state of the server.
apple file type	Perforce file type assigned to Macintosh files that are stored using AppleSingle format, permitting the data fork and resource fork to be stored as a single file.
atomic change transaction	Grouping operations affecting a number of files in a single transaction. If all operations in the transaction succeed, all the files are updated. If any operation in the transaction fails, none of the files are updated.
base	The file revision on which two newer, conflicting file revisions are based.
binary file type	Perforce file type assigned to a non-text file. By default, the contents of each revision are stored in full and the file is stored in compressed format.
branch	(*noun*) A codeline created by copying another codeline, as opposed to a codeline that was created by adding original files. *branch* is often used as a synonym for *branch view*. (*verb*) To create a codeline branch with `p4 integrate`.
branch form	The Perforce form you use to modify a branch.
branch specification	Specifies how a branch is to be created by defining the location of the original codeline and the branch. The branch specification is used by the integration process to create and update branches. Client workspaces, labels, and branch specifications cannot share the same name.
branch view	A specification of the branching relationship between two codelines in the depot. Each branch view has a unique name, and defines how files are mapped from the originating codeline to the target codeline. See *branch*.

Term	Definition
changelist	An atomic change transaction in Perforce. The changes specified in the changelist are not stored in the depot until the changelist is submitted to the depot.
changelist form	The Perforce form you use to modify a changelist.
changelist number	The unique numeric identifier of a changelist.
change review	The process of sending email to users who have registered their interest in changes made to specified files in the depot.
checkpoint	A copy of the underlying server metadata at a particular moment in time. See *metadata*.
client form	The Perforce form you use to define a client workspace.
client name	A name that uniquely identifies the current client workspace.
client root	The root directory of a client workspace. If two or more client workspaces are located on one machine, they cannot share a root directory.
client side	The right-hand side of a mapping within a client view, specifying where the corresponding depot files are located in the client workspace.
client view	A set of mappings that specifies the correspondence between file locations in the depot and the client workspace.
client workspace	Directories on the client computer where you work on file revisions that are managed by Perforce. By default this name is set to the name of the host machine on which the client workspace is located; to override the default name, set the `P4CLIENT` environment variable. Client workspaces, labels, and branch specifications cannot share the same name.
codeline	A set of files that evolve collectively. One codeline can be branched from another, allowing each set of files to evolve separately.

Term	Definition
conflict	One type of conflict occurs when two users open a file for edit. One user submits the file; after which the other user can't submit because of a conflict. The cause of this type of conflict is two users opening the same file. The other type of conflict is when users try to merge one file into another. This type of conflict occurs when the comparison of two files to a common base yields different results, indicating that the files have been changed in different ways. In this case, the merge can't be done automatically and must be done by hand. The type of conflict is caused by non-matching *diff*s. See *file conflict*.
counter	A numeric variable used by Perforce to track changelist numbers in conjunction with the review feature.
default changelist	The changelist used by Perforce commands, unless a numbered changelist is specified. A default pending changelist is created automatically when a file is opened for edit.
default depot	The depot name that is assumed when no name is specified. The default depot name is `depot`.
deleted file	In Perforce, a file with its head revision marked as deleted. Older revisions of the file are still available.
delta	The differences between two files.
depot	A file repository on the Perforce server. It contains all versions of all files ever submitted to the server. There can be multiple depots on a single server.
depot root	The root directory for a depot.
depot side	The left side of any client view mapping, specifying the location of files in a depot.
depot syntax	Perforce syntax for specifying the location of files in the depot.
detached	A client computer that cannot connect to a Perforce server.
diff	A set of lines that don't match when two files are compared. A *conflict* is a pair of unequal diffs between each of two files and a common third file.
donor file	The file from which changes are taken when propagating changes from one file to another.
exclusionary mapping	A view mapping that excludes specific files.

Term	Definition
exclusionary access	A permission that denies access to the specified files.
file conflict	In a three-way file merge, a situation in which two revisions of a file differ from each other and from their base file. Also: an attempt to submit a file that is not an edit of the head revision of the file in the depot; typically occurs when another user opens the file for edit after you have opened the file for edit.
file pattern	Perforce command line syntax that enables you to specify files using wildcards.
file repository	The master copy of all files; shared by all users. In Perforce, this is called the *depot*.
file revision	A specific version of a file within the depot. Each revision is assigned a number, in sequence. Any revision can be accessed in the depot by its revision number, for example: `testfile#3`.
file tree	All the subdirectories and files under a given root directory.
file type	An attribute that determines how Perforce stores and diffs a particular file. Examples of file types are `text` and `binary`.
fix	A job that has been linked to a changelist.
form	Screens displayed by certain Perforce commands. For example, you use the Perforce change form to enter comments about a particular changelist and to verify the affected files.
full-file storage	The method by which Perforce stores revisions of binary files in the depot: every file revision is stored in full. Contrast this with *reverse delta storage*, which Perforce uses for `text` files.
get	An obsolete Perforce term: replaced by *sync*.
group	A list of Perforce users.
have list	The list of file revisions currently in the client workspace.
head revision	The most recent revision of a file within the depot. Because file revisions are numbered sequentially, this revision is the highest-numbered revision of that file.
integrate	To compare two sets of files (for example, two codeline branches) and • determine which changes in one set apply to the other, • determine if the changes have already been propagated, • propagate any outstanding changes.

Term	Definition
Inter-File Branching	Perforce's proprietary branching mechanism.
job	A user-defined unit of work tracked by Perforce. The job template determines what information is tracked. The template can be modified by the Perforce system administrator
job specification	A specification containing the fields and valid values stored for a Perforce job.
job view	A syntax used for searching Perforce jobs.
journal	A file containing a record of every change made to the Perforce server's metadata since the time of the last checkpoint.
journaling	The process of recording changes made to the Perforce server's metadata.
label	A named list of user-specified file revisions.
label view	The view that specifies which file names in the depot can be stored in a particular label.
lazy copy	A method used by Perforce to make internal copies of files without duplicating file content in the depot. Lazy copies minimize the consumption of disk space by storing references to the original file instead of copies of the file.
license file	Ensures that the number of Perforce users on your site does not exceed the number for which you have paid.
list access	A protection level that enables you to run reporting commands but prevents access to the contents of files.
local depot	Any depot located on the current Perforce server.
local syntax	The operating-system-specific syntax for specifying a file name.
lock	A Perforce file lock prevents other clients from submitting the locked file. Files are unlocked with the `p4 unlock` command or submitting the changelist that contains the locked file.
log	Error output from the Perforce server. By default, error output is written to standard error. To specify a log file, set the `P4LOG` environment variable, or use the `p4d -L` flag.

Term	Definition
mapping	A single line in a view, consisting of a left side and a right side that specify the correspondences between files in the depot and files in a client, label, or branch. The left side specifies the depot file and the right side specifies the client files. (See also *client view*, *branch view*, *label view*).
MD5 checksum	The method used by Perforce to verify the integrity of archived files.
merge	The process of combining the contents of two conflicting file revisions into a single file.
merge file	A file generated by Perforce from two conflicting file revisions.
metadata	The data stored by the Perforce server that describes the files in the depot, the current state of client workspaces, protections, users, clients, labels, and branches. Metadata includes all the data stored in the server except for the actual contents of the files.
modification time	The time a file was last changed.
nonexistent revision	A completely empty revision of any file. Syncing to a nonexistent revision of a file removes it from your workspace. An empty file revision created by deleting a file and the `#none` revision specifier are examples of nonexistent file revisions.
numbered changelist	A pending changelist to which Perforce has assigned a number.
open file	A file that you are changing in your client workspace.
owner	The Perforce user who created a particular client, branch, or label.
`P4`	The Perforce Comnand-Line Client program, and the command you issue to execute Perforce commands from the operating system command line.
`P4D`	The program on the Perforce server that manages the depot and the metadata.
P4Diff	A Perforce application that displays the differences between two files. `P4Diff` is the default application used to compare files during the file resolution process.
P4Win	The Perforce Windows Client, a Windows Explorer-style application that enables you to perform Perforce operations and view results graphically.
pending changelist	A changelist that has not been submitted.

Term	Definition
Perforce server	The Perforce depot and metadata on a central host. Also the program that manages the depot and metadata.
protections	The permissions stored in the Perforce server's protections table.
RCS format	Revision Control System format. Used for storing revisions of text files. RCS format uses reverse delta encoding for file storage. Perforce uses RCS format to store text files. See also *reverse delta storage*.
read access	A protection level that enables you to read the contents of files managed by Perforce.
remote depot	A depot located on a server other than the current Perforce server.
reresolve	The process of resolving a file after the file is resolved and before it is submitted
resolve	The process you use to reconcile the differences between two revisions of a file.
resource fork	One fork of a Macintosh file. (Macintosh files are composed of a resource fork and a data fork.) You can store resource forks in Perforce depots as part of an AppleSingle file by using Perforce's `apple` file type.
reverse delta storage	The method that Perforce uses to store revisions of text files. Perforce stores the changes between each revision and its previous revision, plus the full text of the head revision.
revert	To discard the changes you have made to a file in the client workspace.
review access	A special protections level that includes `read` and `list` accesses, and grants permission to run the `review` command.
review daemon	Any daemon process that uses the `p4 review` command. See also *change review*.
revision number	A number indicating which revision of the file is being referred to.
revision range	A range of revision numbers for a specified file, specified as the low and high end of the range. For example, `foo#5,7` specifies revisions 5 through 7 of file `foo`.

Term	Definition
revision specification	A suffix to a filename that specifies a particular revision of that file. Revision specifiers can be revision numbers, change numbers, label names, date/time specifications, or client names.
server	In Perforce, the program that executes the commands sent by client programs. The Perforce server (`p4d`) maintains depot files and metadata describing the files, and tracks the state of client workspaces.
server root	The directory in which the server program stores its metadata and all the shared files. To specify the server root, set the `P4ROOT` environment variable.
status	For a changelist, a value that indicates whether the changelist is new, pending, or submitted. For a job, a value that indicates whether the job is open, closed, or suspended. You can customize job statuses.
submit	To send a pending changelist and changed files to the Perforce server for processing.
subscribe	To register to receive email whenever changelists that affect particular files are submitted.
super access	An access level that gives the user permission to run *every* Perforce command, including commands that set protections, install triggers, or shut down the server for maintenance.
symlink file type	A Perforce file type assigned to UNIX symbolic links. On non-UNIX clients, symlink files are stored as text files.
sync	To copy a file revision (or set of file revisions) from the depot to a client workspace.
target file	The file that receives the changes from the donor file when you are integrating changes between a branched codeline and the original codeline.
text file type	Perforce file type assigned to a file that contains only ASCII text. See also *binary file type*.
theirs	The revision in the depot with which the client file is merged when you resolve a file conflict. When you are working with branched files, *theirs* is the donor file.
three-way merge	The process of combining three file revisions. During a three-way merge, you can identify where conflicting changes have occurred and specify how you want to resolve the conflicts.

Term	Definition
tip revision	In Perforce, the *head revision*. *Tip revision* is a term used by some other SCM systems.
trigger	A script automatically invoked by the Perforce server when changelists are submitted.
two-way merge	The process of combining two file revisions. In a two-way merge, you can see differences between the files but cannot see conflicts.
typemap	A Perforce table in which you assign Perforce file types to files.
user	The identifier that Perforce uses to determine who is performing an operation.
view	A description of the relationship between two sets of files. See *client view*, *label view*, *branch view*.
wildcard	A special character used to match other characters in strings. Perforce wildcards are: • `*` matches anything except a slash • `...` matches anything including slashes • `%d` used for parameter substitution in views
write access	A protection level that enables you to run commands that alter the contents of files in the depot. `Write` access includes `read` and `list` accesses.
yours	The edited version of a file in the client workspace, when you resolve a file. Also, the target file when you integrate a branched file.

Index