Perforce 2005.1 User's Guide

May 2005

Table of Contents

Preface

About This Manual

This is the *Perforce 2005.1 User's Guide*.

This guide documents the Perforce Command-Line Client. Other Perforce client programs, such as P4V, P4Win, and P4Web, are not discussed here. To learn about other Perforce client programs, please see the documentation available on our web site at `http://www.perforce.com`.

The *Perforce User's Guide* uses a tutorial approach to document the commands and flags you're most likely to use when working with Perforce. For complete documentation of every command and every option, consult the *Perforce Command Reference*, or use the built-in command line help system by typing `p4 help`.

Administering Perforce?

If you're responsible for installing and administering a Perforce server, see the *Perforce System Administrator's Guide*. The *System Administrator's Guide* describes how to operate and maintain a Perforce server.

Please Give Us Feedback

We are interested in feedback 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 is a software configuration management (SCM) tool that enables developers to version files, track changes to software development, manage releases, track defects, manage builds, and so on. Specifically:

- 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 copies of files cached on a proxy server.
- Perforce offers *release management* facilities; you can use Perforce to track the file revisions that constitute 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 *lifecycle management* functionality; files can be kept in release branches, development branches, or in any sort of needed file set.
- Perforce supports *change review*; users can be notified by email when particular files are changed.
- Although a build management tool is not built into Perforce, Perforce offers a companion open source product called `Jam`. The `Jam` tool and Perforce are independent of each other; source files managed by Perforce are easily built by `Jam`.

Although Perforce was built to manage source files, you can use Perforce to manage any sort of content, including source code, binary files, other digital assets, HTML pages, formatted documents, or operating system configuration files.

Perforce Server and Perforce Client Programs

Perforce is based on a client/server architecture, in which users at client workstations are connected to a central *server*. Each user works on a client workstation; Perforce client programs on user workstations transfer files between the workstations and the Perforce server. Perforce client programs communicate with the server using TCP/IP.

Workstations running Perforce client programs can be distributed around a local area network, wide area network, dialup network, or any combination thereof. Perforce client programs 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. This program manages the shared file repository and keeps track of users, workspaces, and other Perforce metadata.

  The Perforce Server must run on a UNIX, Mac OS X, or Windows machine.
- Perforce client programs (such as `p4`) run on Perforce client machines. Client programs send user requests to the Perforce Server (`p4d`) for processing, and communicate with `p4d` using TCP/IP.

  Perforce supplies client software for UNIX, Linux, Windows, Mac OS X, and many other platforms.

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

When you use Perforce, you create, edit, and delete files on your own workstation in directories you specify to Perforce as *client workspaces*. You use Perforce commands to move files between the shared file repository (the *depot*) and your local workstation.

When you retrieve files from the depot into your client workspace, you can read, edit, and resubmit the files to the depot to make your changes accessible to other users. When a new *revision* of a file is stored in the depot, the old revisions of the file are preserved and are still accessible.

Files that you edit in your client workspace are aggregated and sent to the depot using *changelists*, which are lists of files and instructions that tell the depot what changes you made to those files. For example, one file might have been edited in your client workspace, another added, and another deleted. These file changes are sent to the depot in a single changelist, which is processed *atomically*: either all the changes are made to the files in the depot, or none are. This approach enables you to simultaneously update all files related to a bug fix or a new feature.

Each client workspace has its own *client workspace view*, which determines what files in the depot are mapped into its owner's 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 development branch. The Perforce Server tracks the state of all client workspaces, including the files in each client workspace, where they reside in the depot, and which files are being worked on by which users.

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 asks the user experiencing the conflict to perform a *resolve* of the conflicting files. The resolve process enables you to decide what needs to be done: should your file overwrite the other user's? Should your own changes be discarded? Or should the two conflicting files be merged into one? At your request, Perforce performs a *three-way merge* between the two conflicting files and the single file that on which both files were based. This process generates a *merge* file from the conflicting files that contains all the changes from both conflicting versions. You can also edit the merged file before submitting the it 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 more about labels, see Chapter 8, Labels.

Branching files

Suppose that one source file needs to evolve in two separate directions; perhaps one set of changes are required for UNIX support, and a second set of changes are required for OS X support. In such cases, two separately-evolving copies of the same files are necessary.

Perforce's *Inter-File Branching*™ mechanism enables you to copy any set of files to a new location in the depot. The new file set, or *codeline*, evolves separately from the old codeline, 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 description of some change that needs to be made to a body of content or 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."

Perforce's job tracking mechanism links jobs to the *changelist*s that implement the job. (A job represents work that is intended to be performed, a changelist represents work actually done.) A job can later be examined to determine if and when it was fixed, what files were modified to implement the fix, who fixed it, and whether the fix was 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. Perforce's job tracking functionality can be used as is, or you can integrate Perforce jobs with third-party job tracking systems by using P4DTI - Perforce Defect Tracking and Integration.

For more about jobs, please see Chapter 10, Job Tracking.

Change review

Perforce's *change review* mechanism enables 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 in Perforce is implemented by an external program, or *daemon*, which can be customized.

Perforce can be made to run external scripts when changelists are submitted or forms are changed. These scripts, called *triggers*, enable you to validate changelists and forms, start build processes or perform other tasks related to workflow.

To learn how to set up the change review daemon, integrate Perforce with third-party defect tracking systems, or develop your own custom daemons and triggers, see 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 which Perforce commands can be run by any particular user. Permissions can be granted or denied based on usernames, workstation IP addresses, and administrator-defined user groups.

Perforce protections are discussed in the *Perforce System Administrator's Guide.*

Other Perforce Client Programs

The Perforce Command-Line Client (`p4`) is not the only Perforce client program. Other Perforce client programs are available on the Perforce web site, including P4V, P4Win, and P4Web.

P4V

The Perforce Visual Client (P4V) is a graphical user interface to Perforce on Mac OS X, UNIX, Linux, and Windows. P4V provides quick and easy access to Perforce-managed files through a user interface that is consistent across many operating systems.

For more about P4V, see the product page at:

`http://www.perforce.com/perforce/products/p4v.html`

P4Win

The Perforce Windows Client (P4Win) provides a native Microsoft Windows user interface for all SCM tasks. Using the familiar Windows Explorer look and feel, P4Win shows your work in progress at a glance and provides easy access to Perforce tasks.

For more about P4Win, see the product page at:

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

P4Web

The Perforce Web Client (P4Web) turns any Web browser into a complete SCM tool. P4Web works with a Perforce Server at Release 99.2 or higher, and runs on UNIX, Linux, Mac OS X, and Windows platforms.

For more about P4Web, see the product page at:

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

Merge Tools

Interactive merge tools enable 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).

P4V

The Perforce Visual Client provides built-in merge capability for Mac OS X, UNIX, Linux, and Windows.

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 familiar three-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. You need only change an environment variable (such as `P4MERGE` or `P4DIFF`) to point to your 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 already have 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, what work that was done to resolve an issue, or to generate reports relating issues to files or codelines.

Activity in your Perforce depot - such as 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 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 and can be used with a Perforce Server on any platform at Release 2000.2 or higher. 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 enable developers to work with Perforce from within integrated development environments (IDEs) such as Visual Studio .NET, JBuilder, Eclipse, WebSphere Studio, CodeWarrior, and many more.

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 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 applications to work with documents under Perforce control.

For more about P4OFC, see:

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

P4EXP

The Perforce Plug-in for Windows Explorer (P4EXP) enables one-click access to Perforce operations (including add, edit, diff, view revision history, and more) from within the familiar Windows Explorer environment.

For more about P4EXP, see:

```
http://www.perforce.com/perforce/products/p4exp.html
```

Chapter 2

Connecting to the Perforce Server

Perforce is based on a client/server architecture. Users work on files in client workspace directories on their own machines; these files are transferred to and from a shared file repository located on a Perforce server. Every Perforce system uses one server and can have many client workstations.

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

- The Perforce Server (`p4d`) runs on the Perforce server machine, manages the shared file repository, and keeps track of users, workspaces, and other Perforce metadata.
- Perforce client programs (such as `p4`) run on Perforce client machines, sending user requests to the Perforce Server (`p4d`) for processing.

Perforce client programs use TCP/IP to communicate with the Perforce Server. To use Perforce, you must supply your client program with the address and port of the Perforce server to which you want to connect. See "Setting up your environment to use Perforce" on page 21 for details.

Before you begin

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

If this is not the case (for instance, if you're installing Perforce for the first time), you must install the Perforce server before continuing. For an overview of how to set up a server, see "Installing Perforce" on page 143.

The installation information in this manual 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, read the full installation instructions in the *System Administrator's Guide*.

Setting up your environment to use Perforce

In order to connect to a Perforce server, you must supply your Perforce client program with two pieces of information:

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

To connect to a Perforce server, specify the host and port number by setting the `P4PORT` environment variable to *`host:port`*, where *`host`* is the name of the host on which the `p4d` server is running, and *`port`* is the port on which the `p4d` server 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`

For information about setting environment variables for most operating systems and shells, see "Setting and viewing environment variables" on page 148.

If your site is already using Perforce, your system administrator might have already set `P4PORT` for you; if not, you'll need to set `P4PORT` yourself.

- If you've just installed the Perforce server yourself, you already know the host and port number, having configured the server on a specific host to listen to a specific port.
- If you're connecting to an existing Perforce installation, ask your system administrator for the host and port of the Perforce server. By default, a Perforce server listens on port 1666.

After you have set `P4PORT` to point to your server, test your connection to the Perforce server by using the `p4 info` command. See "Verifying the connection to the Perforce server" on page 23.

If your Perforce client program is running on the same host as the server, you can omit the host and specify only the port number. If the Perforce server is listening to the default port 1666, you need only specify the host name in `P4PORT`. If the Perforce server is running on a host named or aliased `perforce`, and is listening on the default 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>`*

Verifying the connection to the Perforce server

To verify your connection to the Perforce server, enter `p4 info` at the command line. If the `P4PORT` environment variable is set correctly, 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: 2004/06/10 12:11:47 -0700 PDT
Server version: P4D/FREEBSD/2004.2/62360 (2004/06/10)
Server license: P4 Admin <p4adm> 20 users (expires 2005/01/01)
```

The `Server address:` field shows the Perforce server to which `p4` has connected, as well as the host and port number on which `p4d` is listening.

In the above example, the connection was successfully tested. If you receive an error message like this:

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

then the `p4` client program failed to connect to `p4d`, either because the Perforce server wasn't running, or because your setting for `P4PORT` was incorrect.

If the value shown in the third line of the error message is `perforce:1666` (as above), then `P4PORT` was unset; if the value is anything else, `P4PORT` was incorrectly set. In either case, you must change the value of `P4PORT` to point to your Perforce server.

Windows On Windows platforms, use the command `p4 set` to set registry keys, rather than environment variables.

For instance, to connect to the server in the example above, use the command `p4 set P4PORT=p4server:1818` from the Command Prompt.

Logging in to Perforce

Depending on how your system administrator has defined your server's security policy, you may need to log in to Perforce before you can run Perforce commands. For more about Perforce passwords and authentication, see "Perforce Passwords" on page 79.

Perforce system administrators should consult the *System Administrator's Guide* for details on how to determine what type of authentication is best for the users at your site.

Chapter 3 Perforce Basics: Quick Start

This chapter teaches basic Perforce usage. You'll learn how to set up your 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 use some 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

Working in Perforce is simple: you create, edit, and work on files a set of directories on your local machine. These directories make up your *client workspace*. You use Perforce commands to move files to and from the shared file repository, called the *depot*. Other Perforce users retrieve files from the depot into their own client workspaces, where they can read, edit, and submit their changes 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 few changes to your normal work habits are required. You work on files in your own directories using your editor or IDE of choice; Perforce commands supplement your normal work actions instead of replacing them.

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

Note Many `p4` commands display a form for editing in a standard text editor. You can specify your text editor of choice by setting the `P4EDITOR` environment variable.

Setting up a client workspace

To move files between the depot and a client workspace, 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 your 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 a project called Elm. He wants to refer to the set of files he's working on (his client workspace) by the name `eds_elm`*. In the Korn or Bourne shells, he types:*

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

Subsequent `p4` *commands will use the client workspace named* `eds_elm`*.*

Different operating systems and shell have their own methods of setting environment variables. See "Setting and viewing environment variables" on page 148 for details.

Describing your client workspace to the Perforce server

To define the new client workspace that you specified by setting `P4CLIENT`, or to edit an existing client workspace, use the `p4 client` command. Enter:

```
p4 client
```

Typing `p4 client` brings up the client definition form in a text editor. After you have filled out the form and exited the text editor, you can use Perforce client programs to move files between the depot and your client workspace.

The `p4 client` form contains a number of fields; at this point, the important fields are the `Root:` and `View:` fields:

Field	Meaning
`Root:`	Your *client workspace root* is the topmost subdirectory of your client workspace. Set your client workspace root to a directory under your control; you will be doing most of your development work on files located in your client workspace.
`View:`	The *client workspace view* determines which files and directories in the depot are mapped to your client workspace, and where the files appear beneath your client workspace root.

Note In the text forms used by Perforce, field names always start in the leftmost column of text, and field values are indented with tabs or spaces. Perforce requires that there be at least one space or a tab prior to the contents of a form field.

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

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

```
Client: eds_elm
Owner:  edk
Description:
        Created by edk.
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) are mapped to the depot, and all files in the depot are mapped to Ed's home directory, likely cluttering it with files that Ed isn't interested in seeing.*

Ed chooses to work on all Elm-related material in an `/elm` *subdirectory beneath his home directory of* `/usr/edk`*, and he would like this directory (*`/usr/edk/elm`*) to contain only files in the* `elm_proj` *portion of the depot. He changes the values in the* `Root:` *and* `View:` *fields as follows:*

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

The revised form specifies `/usr/edk/elm` *as the top level directory of Ed's client workspace, and that the files in 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.*

In the `p4 client` form, the read-only `Client:` field contains the string stored in the `P4CLIENT` environment variable. The `Description:` is a free-form text field where you can add a description of the workspace. The `View:` field describes the relationship between files in the depot and files in the client workspace. To use Perforce properly, it is crucial to understand how views work. Views are discussed in greater detail in "Mapping Depot files to your Client Workspace" on page 38.

Creating a client specification has no immediate visible effect; no files are created when a client specification is created or edited. The client workspace specification merely defines where files are located when you populate your workspace with files from the server.

Copying depot files into your workspace

To retrieve files from the depot into a client workspace, use the `p4 sync` command.

Note If you've just installed a Perforce server for the first time, `p4 sync` won't do anything, because an empty depot contains no files to copy to client workspaces.

If you have just installed a Perforce server, use `p4 add`, as described in "Adding files to the depot" on page 29, to populate the depot by copying files from your client workspace to the depot.

The `p4 sync` command maps depot files through your client workspace view, compares the result against the contents of your client workspace, and then adds, updates, or deletes files in your workspace as needed to bring the contents of your workspace into sync with the depot.

If a file exists within a particular subdirectory in the depot, but that directory does not yet exist in your client workspace, the directory is created in the client workspace when you run `p4 sync`. If a file in your client workspace has been deleted from the depot, `p4 sync` removes the file from your client workspace.

By default, `p4 sync` updates your entire client workspace. To limit the update to only a portion of your client workspace, you can supply filenames or wildcards to `p4 sync`.

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

Lisa is responsible for the documentation for the Elm project. Her client workspace root is a directory called `elm_ws`*, and she has set her client workspace view to include only the* `elm_proj` *documentation tree. To populate her client workspace, she enters her client workspace root, runs* `p4 sync`*, and sees:*

```
$ 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.>
```

After the `p4 sync` *command completes, the most recent revisions of the Elm project's documentation files as mapped through Lisa's client workspace view, are available in her workspace.*

Updating the depot with files from your workspace

Any file in your client workspace can be added to, updated in, or deleted from the depot in a two-step process:

1. You add information about changes to files in your client workspace to a *changelist* by using the commands `p4 add` *`filenames`*, `p4 edit` *`filenames`*, or `p4 delete` *`filenames`*. A Perforce changelist is a list of files and operations (such as adding a new file to the depot, editing an existing file, or deleting a file from the depot) to be performed in the depot.
2. You use the `p4 submit` command to commit your changes to the depot. When the `p4 submit` command successfully completes, the changes to the files in your workspace are reflected in the depot.

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 changes to the files in the depot occur only after the changelist is submitted to the depot with `p4 submit`.

Changelists enable you to send updated sets of files to the depot in indivisible, or *atomic*, transactions: when a changelist is submitted, either all of the files in the changelist are committed, or none of the files are committed.

Note This chapter discusses only the *default changelist*, which is automatically maintained by Perforce.

Changelists are a key concept in Perforce. For a full discussion, see Chapter 7, Changelists.

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* (for add, for edit, or for delete) 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 includes the files in the default changelist.

After you have added files to the changelist, you must submit your changelist to the depot by using the `p4 submit` command. See “Submitting your changes to the depot” on page 32 for details.

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, four files have been added to his default changelist. The files are not actually stored in the depot until Ed uses the `p4 submit` *command to submit the changelist.*

In the example shown, the filenames are displayed as *`filename`*`#1`. The `#`*`n`* suffix is used by Perforce to indicate the *n*th revision of this file. The first revision of any file is revision `#1`. Revision numbers are always assigned sequentially.

Adding more than one group of files at once

You can supply multiple file arguments on the command line. For example:

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.>
```

Files in the three specified directories are added to the default changelist.

Populating an empty depot

In Perforce, there is no difference between adding files to an empty depot and adding files to a depot that already contains other files. You can populate a new, empty depot 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 the `p4 edit` command to open the file for `edit` in the default changelist. Opening a file for edit has three effects:

- Write permissions are turned on in your client workspace for the file(s) being edited, enabling you to change the copy of the file residing in your workspace.
- The Perforce server records the fact that you are working on a file, so that other users are aware that someone else is working on the file.
- The file(s) you are editing are added to your default changelist, so that you can submit your work when you are done.

After you have opened a file for edit and made the required changes in your client workspace, use the `p4 submit` command make your changes available to other users by submitting the changelist. See “Submitting your changes to the depot” on page 32.

Note Before you can open a file for `edit`, someone must have already added the file to the depot with `p4 add`, or copied into your 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
```

Ed then edits the file with his preferred text editor. When he’s finished making his changes to the file, he makes his changes available to other users by using `p4 submit` *to submit the changelist to the depot.*

Deleting files from the depot

To delete a file from the depot, use the `p4 delete` command to open the file for `delete` in the default changelist.

The `p4 delete` command deletes the file from your client workspace as soon as you run the `p4 delete` command, but deletion of the file in the depot does not occur until you use `p4 submit` to submit the changelist containing the delete operation to the depot.

After you submit a changelist with a file deletion, it appears to all users as though the file is deleted from the depot. Actually, only the most recent revision (or *head* revision) of the file is marked as deleted. Older revisions of the file are not removed, and by specifying these older revisions, you can always recover old revisions of “deleted” files back into your client workspace.

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 Ed's client workspace immediately, but it is not deleted from the depot until Ed submits the changelist with the `p4 submit` *command.*

Submitting your changes to the depot

In Perforce, *changelists* are used to group related changes (for example, a bug fix, or the addition of a new feature to a product) in a logical fashion. Most changelists contain more than one file. None of the operations you perform on files (such as opening for add, edit, or delete) take effect in the depot until you commit your changes to the depot by *submitting the changelist.*

Submitting a changelist to the depot works atomically: either all the files in the changelist are updated in the depot, or none of them are. (In Perforce terminology, this is called an *atomic change transaction*).

To submit a changelist, use the `p4 submit` command.

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

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

Ed adds the new file and prepares to edit the existing files as follows:

```
$ 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 header file and the documentation files in his workspace. When he is ready to submit the changelist, he types `p4 submit` *and sees the following form in a standard text editor:*

```
Change: new
Client: eds_elm
User:   edk
Status: new
Description:
        <enter description here>
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
```

Ed changes the contents of the `Description:` *field in the* `p4 submit` *form to describe the changes he's made as follows:*

```
Change: new
Client: eds_elm
User:   edk
Status: new
Description:
        Changes to Elm's 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 Ed's changes 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.

The `p4 submit` form includes a `Description:` field; you must supply a description in order for your changelist to be accepted.

The `Files:` field contains the list of files in the changelist. If you remove files from this field, any files not included in the changelist are carried over to a new default changelist, and reappear the next time you use `p4 submit`.

If a directory containing a new file does not exist in the depot, the directory is automatically created in the depot when you submit the changelist.

When you run `p4 submit`, the operating system's write permission on the submitted files in your client workspace is turned off. Write permissions are restored when you open a file for editing with `p4 edit`.

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

See Chapter 5, Perforce Basics: Resolving File Conflicts for details on submitting numbered changelists.

Backing out: reverting files to their unopened states

To back out unwanted changes made for files that you opened for `add`, `edit`, or `delete`, use the `p4 revert` command. This command restores the file in the client workspace 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 plans to change, and that he did not intend to open it. He reverts* `signals.c` *to its unopened state with* `p4 revert`*:*

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

If you use `p4 revert` on a file you opened with `p4 delete`, the file reappears in your client workspace immediately. If you revert a file opened with `p4 add`, the file is removed from the changelist, but is left intact in your client workspace.

If you revert a file you originally opened with `p4 edit`, the last synced version is written back to the client workspace, overwriting the edited version of the file. To reduce the risk of accidentally overwriting your changes, you can preview any 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 many 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.
`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 licensing information.

Two other reporting commands are useful when getting started with Perforce:

Command	**Meaning**
`p4 have`	Lists all file revisions synced to your client workspace.
`p4 sync -n`	Report the set of files that would be updated in the client workspace by a `p4 sync` command 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, see Chapter 3, Perforce Basics: Quick Start.

What is a Client Workspace?

A Perforce client workspace is a collection of files on a user's workstation that are managed by a Perforce client program. You can have more than one Perforce client workspace per workstation. The Perforce server tracks the state of all client workspaces owned by all users, including what files are on which users' workspaces and which users are working on which files.

Every client workspace on every user's workstation has a unique name that identifies the workspace to the Perforce server. If you do not specify a name for your client workspace, the name defaults to your workstation's name. You can override the default value by setting the `P4CLIENT` environment variable.

All files within a client workspace share a common root directory called the *client workspace root*. The client workspace root is the highest-level directory on your workspace under which the managed source files reside. Files under your client workspace root that are not managed by Perforce are ignored by Perforce client programs, enabling you to use Perforce to manage the source files in your client workspace while ignoring non-versioned files such as compiled object files and other temporary files created by your development tools.

Perforce client programs manage versioned files in your client workspace in three ways:

- When you use `p4 sync` to synchronize your client workspace with the depot, your client program creates, updates, and deletes files in your workspace as required
- When you open a file for editing with `p4 edit`, your client program restores write permission to the copy of the file in your client workspace.
- When you submit your changes back to the depot with `p4 submit`, your client program transmits your changes back to the depot and deactivates write permission on your local copy of the file.

It's possible to circumvent Perforce by manually altering permissions on files in your workspace, but using Perforce is easier than circumventing it. For instance, if you need to make a temporary change to a file in your workspace, it is easier to use Perforce to open the file for edit, make your change, and then use `p4 revert` to discard your change, rather than to manually override file permissions or copy and restore the file.

To avoid confusion caused by inadvertent changes to workspace files, Perforce client programs include commands to verify that the state of your client workspace matches the Perforce server's record of your workspace state.

Mapping Depot files to your Client Workspace

The Perforce server manages the depot - the central repository containing every revision of every file under Perforce control. Depot files are stored in folder hierarchies, rather like those on a large hard drive.

To control where depot files appear under your client workspace root, you must map the files and directories on the Perforce server to the corresponding areas of your client workspace. These mappings constitute your *client workspace view*.

Setting up a client workspace view doesn't transfer any files from the server to your computer. The view only sets up the mapping that controls the relationship between the depot and your client workspace when files are transferred.

You configure your client workspace view with `p4 client` command. When you use the `p4 client` command, Perforce displays a form similar to this one:

```
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 files in the depot appear on your workstation when you use `p4 sync` to synchronize your workspace, and where files in your client workspace are stored in the depot when you use `p4 submit` to update the depot with your changes.

Note The `p4 client` form has more fields than those described here. For a complete list, see the *Perforce Command Reference*.

Client workspace views

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

```
//depotname/file_specification
```

The right-hand side of each mapping specifies one or more files or directories in a 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*.

All client views, regardless of how many mappings they contain, perform the same two functions:

- *Define the set of files in the depot that can appear in a client workspace.*

  The files that appear in a client workspace are determined by the sum of the depot sides of the mappings within a view. A view might include every file in the depot in a client workspace, only the files within two directories, or even 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. Regardless of how many lines there are in a client view, there is always a one-to-one mapping between depot files and client workspace files.

The one-to-one mapping can be straightforward. For instance, the default view in the `p4 client` form is quite simple:

```
//depot/...    //clientname/...
```

This mapping maps the entire depot to the entire client workspace; you can edit the default mapping to include only the portion of the depot with which you're working.

More complex mappings are possible. For instance, you can have files in the depot appear in different subdirectories in your client workspace, or even rename files by configuring the mapping. For a complete list of ways to control how and where the depot files appear in your workspace, see "Types of mappings used in views" on page 40.

To determine the exact location of any client file on a workstation, 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 workspace `eds_elm` is set to `/usr/edk/elm`, then the depot file `//eds_elm/doc/elmdoc.1` appears in `/usr/edk/elm/doc/elmdoc.1` whenever the `eds_elm` workspace is synced.

Using wildcards in views

Perforce uses three wildcards for pattern matching on the command line and in views. Any combination of these wildcards can be used together.

Wildcard	Meaning
`*`	Matches anything except slashes; matches files in a single directory.
`...`	Matches anything including slashes; matches in the current directory all subdirectories.
`%%d`	Used for parametric substitution in views. See "Changing the order of filename substrings" on page 42 for a full explanation.

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 used in views

To control the mapping between depot files and your client workspace, set up your client workspace view by editing the `View:` field in the `p4 client` form. You can configure your client workspace to contain only the subset of files you're interested in, map files in one depot subdirectory to different subdirectories in your workspace, name files differently in the depot and your client workspace, and even map in files from other depots.

Direct client-to-depot views

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

```
//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. For a large site, such a view is inconvenient, as most users only need to see a small subset of the files in the depot.

Including only part of the depot in your client workspace

To have only the portion of the depot in which you're interested appear in your client workspace, change the left-hand side of the `View:` field to include only the relevant portions 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 workspace

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 "When two mappings conflict" on page 43 for details.

Excluding files and directories from the view

Exclusionary mappings enable you to exclude files and directories from a client workspace by prefacing the mapping with a minus sign (-). Whitespace is not permitted 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/...
```

Because later mappings have precedence over earlier ones, no files from the depot's `doc` *subdirectory are copied into Bill's workspace. Conversely, if Bill has a* `doc` *subdirectory in his client, no files from that subdirectory are ever 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. He uses the following mappings:*

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

When two mappings conflict

When you use multiple mappings in a single view, some files can map to two separate places in the depot or on the client. When two mappings conflict in this way, the later mapping overrides the earlier mapping.

Example: *Mappings that conflict.*

Joe has constructed a view as follows:

```
View:
        //depot/proj1/...    //joe/project/...
        //depot/proj2/...    //joe/project/...
```

The second mapping `//depot/proj2/...` *maps to* `//joe/project`*, and conflicts with the first mapping. Because these mappings conflict, the first mapping is ignored; no files in* `//depot/proj1` *are mapped into the workspace:* `//depot/proj1/file.c` *is not mapped, even if* `//depot/proj2/file.c` *does not exist.*

Overlaying multiple mappings into one workspace

Overlay mappings enable you to map files from more than one depot directory in the same place in a client workspace. To overlay the contents of a second directory in your client workspace, use a + in front of the mapping.

Example: *Overlaying multiple directories in the same workspace.*

Joe has constructed a view as follows:

```
View:
        //depot/proj1/...    //joe/project/...
        +//depot/proj2/...    //joe/project/...
```

The overlay mapping `+//depot/proj2/...` *maps to* `//joe/project`*, and overlays the first mapping. Overlay mappings do not conflict. Files in* `//depot/proj2` *take precedence over files in* `//depot/proj1`*: if* `//depot/proj2/file.c` *is missing,* `//depot/proj1/file.c` *is mapped into the workspace instead.*

Overlay mappings are often useful for applying sparse patches within the context of build environments.

Mapping Windows workspaces across multiple drives

To specify a Perforce client workspace that spans multiple Windows drives, use a `Root:` of `null`, and specify the drive letters in the client workspace view. Use uppercase drive letters when specifying workspaces across multiple drives. For example:

```
Client: eds_win
Owner:  edk
Description:
         Ed’s Windows Workspace
Root:   null
Options:        nomodtime noclobber
View:
        //depot/main/...     "//eds_win/C:/Current Release/..."
        //depot/rel1.0/...   //eds_win/D:/old/rel1.0/...
        //depot/rel2.0/...   //eds_win/D:/old/rel2.0/...
```

Mappings that include files from multiple depots

By default, each Perforce server contains a single depot, and the name of the depot is `depot`. Perforce servers can be configured to use multiple depots. If your administrator has configured more than one depot on your server, the default client workspace form looks like this:

```
View:
        //depot/...         //eds_elm/depot/...
        //testing/...       //eds_elm/testing/...
        //archive/...       //eds_elm/archive/...
```

If your system administrator has created multiple depots on your server, you can include files from more than one depot in the same client workspace. The rules for setting up client workspaces that map files from multiple depots are the same as those for client workspace views that map files from a single depot.

See the *System Administrator’s Guide* for information on how to configure multiple depots on a Perforce server.

Client Workspace Specification Options

To change your client workspace specification, use the `p4 client` command. The `Options:` field in the `p4 client` form contains six values, separated by spaces. Each of the six options has two possible settings:

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`

Changing workspace views or moving your workspace root

You can use `p4 client` to change your workspace specification at any time, but changes to client workspace specifications do not take effect when you use `p4 client`. Your changes to a workspace specification take effect only when you *use* that specification after having updated it.

Because client workspace changes take effect only after you use a changed client specification, changing your workspace view or root can sometimes lead to confusing behavior. To avoid confusion:

- If you change your client workspace `View:` field with `p4 client`, perform a `p4 sync` immediately after doing so. The files in your client workspace will then be removed from their old locations and written to their new locations.
- If you use `p4 client` to move your client workspace `Root:`, use `p4 sync #none` to remove versioned files from their old location in your workspace *before* using `p4 client` to change the client root. After you have used `p4 client` to change the client root, 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 old client workspace locations manually).
- Avoid changing your workspace root and client view at the same time. Change either the `Root:` or the `View:` field, perform a `sync` to ensure that the files are in place in your new client workspace (or removed from your old workspace), and then change the other field.

Configuring line-ending conventions (CR/LF translation)

Use the `LineEnd:` field to define what translation (if any) of line-ending character(s) takes place when transferring text files to and from the depot and your 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

Option	Meaning
`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. In shared mode, when you sync your client workspace, line endings will be `LF`. If you edit a text file on a Windows machine, and your editor inserts `CR`s before each `LF`, the extra `CR`s are removed upon file submission and do not appear in the archived file. The most common use of the `share` option is for users of Windows workstations who mount their UNIX home directories as network drives; if you sync files onto a UNIX directory mounted as a network drive and edit your files on a Windows workstation, the `share` option eliminates problems caused by Windows-based editors' insertion of carriage returns in text files.

Multiple workspace roots for cross-platform work

If you are working on more than one operating system, but want to use the same client workspace for each machine, use the `AltRoots:` field in the `p4 client` form to specify up to two alternate client workspace root directories.

Note If you are using a Windows directory in any of your client workspace roots, you must specify the Windows directory as your main workspace `Root:` and your other workspace root directories in the `AltRoots:` field.

If you have any `AltRoots:` configured, your Perforce client program compares the current working directory against the main `Root:` first, and then against the alternate roots, and uses the first root that matches the current working directory as the workspace root for that command. If no roots match, the main workspace root is used.

For example, if `edk`'s current working directory is under `/usr/edk/elm`, then Perforce uses the UNIX path as his client workspace root, rather than `e:\porting\edk\elm`, enabling `edk` to use the same workspace specification for UNIX and Windows work:

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

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

Deleting a client workspace specification

Use `p4 client -d` *`clientname`* to delete a client workspace specification. 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 free up disk space on your local workstation by removing versioned files from an old client workspace, either use `p4 sync #none` (described in "Specifying File Revisions" on page 53) on the files *before* deleting the client specification, or delete the files manually using your operating system's file deletion commands *after* deleting the client specification.

Referring to Files on the Command Line

When you provide file names as arguments to Perforce commands, you can do so either by using the names of the files as they exist in your client workspace (*local syntax*), or by using Perforce's cross-platform syntax (*Perforce syntax*).

Local syntax and Perforce syntax

To use *local syntax*, specify files as you would in your local operating system shell. You can specify file names as absolute paths or relative to your current working directory. If you use relative path components, provide relative paths at the beginning of the file name (that is, `./dir/file.c` is allowed, but `dir/../dir/file.c` is not allowed).

To use *Perforce syntax*, specify files relative to either a client workspace root ("client syntax"), or the top of the depot tree ("depot syntax"). Unlike local syntax, which uses the conventions of your operating system or command shell, Perforce syntax is identical across different operating systems.

Filenames specified in Perforce syntax always begin with two slashes (//), followed by the client workspace or depot name, followed by the full path of the file relative to the client workspace root or the top of the depot tree. Path components in client and depot syntax are always separated by forward slashes (/), regardless of the component separator used by the local operating system.

The following table shows how to use both forms of Perforce syntax, as well as one form of local syntax, to specify the same file:

Syntax	**Example**
Depot syntax	`//depot/main/src/module/file.c`
Client syntax	`//myworkspace/module/file.c`
Local syntax	`C:\Projects\working\module\file.c`

Using local syntax and Perforce syntax on the command line

You can use any combination of client syntax, depot syntax, or local syntax to specify files to Perforce commands. Your Perforce client program evaluates any necessary mappings to determine which file is actually used.

For instance, if you supply a filename in client syntax or local syntax, your Perforce client program uses your client workspace view to locate the corresponding file in the depot. If you refer to a filename using depot syntax, your Perforce client program uses your client workspace view to locate the corresponding file in the client workspace.

Client workspace names and depot names on the same Perforce server share the same namespace; it is impossible for a client workspace name to be the same as a depot name.

Example: *Using different syntax forms to refer to a file.*

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

From 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 his workstation, 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
```

You can provide filenames to any Perforce command in client syntax, depot syntax, or local syntax. The examples in this manual use all three syntax forms interchangeably.

Using wildcards in Perforce commands and views

Perforce wildcards can be used in both local and 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 in the current directory and its subdirectories
`....c`	All files in the current directory and its subdirectories ending in `.c`
`/usr/edk/...`	All files in `/usr/edk` and its subdirectories.
`//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)

Perforce wildcards and the command line

The `p4` command passes "`...`" wildcards directly to the Perforce server, where the server expands the wildcard to match against files available on the server. Most command shells ignore the `...` and `%%d` wildcards, but expand the `*` wildcard, passing the files that match the `*` as multiple arguments to the `p4` command. To have Perforce match the `*` wildcard against the contents of the depot, escape the `*` wildcard with quotation marks, backslashes, or whatever convention is used by your shell.

To add multiple files with `p4 add`, use the `*` wildcard. You cannot use the "`...`" wildcard with `p4 add` to add all files and subdirectories beneath a directory, because the "`...`" wildcard is expanded by the Perforce server: as files being added do not exist on the server, the server has no means of expanding the "`...`" wildcard when it is used with the `p4 add` command. The `*` wildcard is expanded by the local OS shell, not by the Perforce server, and works with `p4 add`.

Name and string limitations for filenames and Perforce objects

The pathname component separator (`/`) and recursive subdirectory wildcards (`...`) are not permitted in file names, label names, or other identifiers.

Character	Reason
`...`	Perforce wildcard: matches anything, works at the current directory level and includes files in all directory levels below the current level.
`/`	Perforce separator for pathname components.

To refer to files containing the Perforce revision specifier wildcards (`@` and `#`), file matching wildcard (`*`), or positional substitution wildcard (`%%`) in either the file name or any directory component, use the ASCII expression of the character's hexadecimal value. ASCII expansion applies only to the following four characters:

Character	ASCII expansion
`@`	%40
`#`	%23
`*`	%2A
`%`	%25

To add a file such as `status@june.txt`, force a literal interpretation of special characters by using:

```
    p4 add -f //depot/path/status@june.txt
```

When you submit the changelist, the characters are automatically expanded and appear in the change submission form as follows:

```
    //depot/path/status%40june.txt
```

After submitting the changelist with the file's addition, you must use the ASCII expansion in order to sync the file to your workspace or edit it within your workspace. For example:

```
    p4 sync //depot/path/status%40june.txt
    p4 edit //depot/path/status%40june.txt
```

Most special 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.

Using spaces in file and path names

Use quotation marks to enclose depot-side or client side mappings of files or directories that contain spaces. For instance, the mapping:

```
    "//depot/release 1.2/doc/..." "//eds_ws/1.2 documentation/..."
```

maps all files in `//depot/main/release 1.2/doc` into the `1.2 documentation` subdirectory of client workspace `eds_ws`.

Other Perforce objects, such as branch names, client names, label names, and so on, can be *specified* with spaces, but these spaces are 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, can be of any length. All names given to Perforce objects such as branches, clients, and so on, are limited to 1024 characters.

Internationalization and non-ASCII characters in filenames

In order to support internationalization, Perforce permits the use of "unprintable" (that is, non-ASCII) characters in filenames, label names, client workspace names, and other identifiers. Although non-ASCII characters are permitted in filenames and Perforce identifiers, entering such characters on a command line might require platform-specific solutions. Users of GUI-based file managers can manipulate such files with drag-and-drop operations.

In internationalized environments, there are additional limitations on how Perforce client programs display filenames in non-ASCII character sets. To ensure that all filenames are displayed consistently across all localized machines in an organization using internationalized character sets, all users must use a common code page setting (under Windows, use the **Regional Settings** applet in the **Control Panel**; under UNIX, set the `LOCALE` environment variable).

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

Specifying File Revisions

Perforce uses the # character to identify file revisions. File revisions in Perforce are denoted by sequentially-increasing integers, beginning from #1 for the first revision, and so on.

The most recent revision of a file is the highest-numbered revision on the server, and is called the *head revision*. The revision you last synced to your workspace is called the *have revision*. The zeroth revision of a file is called the *null revision*, and contains no data.

Specifying file revisions with filenames

All of the commands and examples shown so far have been used against the most recent version of the files to which they apply, but many Perforce commands can act on older file versions.

For instance, if you type `p4 sync //workspace/src/lock.c`, the latest revision, or *head revision*, of `lock.c` is retrieved into your workspace, but you can use revision specifiers against `lock.c` to obtain the first revision, the revision as of a certain date and time, or the revision of the file as of the submission of a changelist number.

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``@`*`changenum`*	A change number	`p4 sync lock.c@126`
		Refers to the version of `lock.c` when changelist 126 was submitted, even if no changes to `lock.c` were submitted in changelist 126.
		The file specification used in a command like
		`p4 sync //depot/...@126`
		refers to the entire depot (`//depot/...`) as of the changes in 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).
`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.

Revision Specifier	Meaning	Examples
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.

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.
		Use four digits when specifying years. If you use dates with two-digit years, the year is assumed to be in the twentieth century.

Date and time specifications are always interpreted in terms of the local time zone of the Perforce server. Because the server represents timestamps 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 be correct in the new timezone.

You can discover the date, time, offset from GMT, and time zone in effect at your Perforce server by examining the "`Server date:`" line in the output of `p4 info`.

Specifying file revisions without filenames

Revision specifications can be provided without file names. If you do not specify a filename with a revision specifier, the command is assumed to apply to the specified revision of all files in the depot or in the client's workspace. For instance, `#head` refers to the head revisions of every file in the depot, and `@labelname` refers to the revisions of every file tagged with the label *labelname*.

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
```

Lisa wants to populate her client workspace with the file revisions that Ed last synced to his workspace. She types:

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

All files in `//depot/elm_proj/doc/...` *are synced to Lisa's workspace at the revisions at which they were last synced to Ed's workspace: in this case, as of the submission of changelist 30.*

Example: *Removing files from a client workspace.*

Ed wants to remove all Perforce-controlled files from his client workspace. He types:

```
p4 sync #none
```

All files in his client workspace that are managed by Perforce are removed from his workspace ("synced to the nonexistent revision"), but are not removed from the depot.

Specifying ranges of revisions

Some Perforce commands can be applied to a range of revisions, rather than just a single revision. A revision range is two revision specifications, separated by a comma.

The commands that accept revision range specifications are:

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

If you provide a revision specifier where a revision range is expected, the specified revision is assumed to be the end of the revision range, and the start of the revision range is assumed to be revision #1. If no revision number or range is given where a revision range is expected, all revisions are assumed (that is, revision #1 through #head).

Example: *Listing changes with revision ranges.*

A release manager needs to see a list of all changes to the Elm project in July, and 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'
```

The manager can then use `p4 describe` *`change`* *against any desired changelist number to obtain a full description.*

Perforce 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 can be applied to the base types to enable preservation of timestamps, support for RCS keyword expansion, file compression on the server, and more.

When you add a new file to the depot, Perforce attempts to automatically determine the type of the file. For instance, you use `p4 add` on a new file, your Perforce client program determines whether or not 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 unicode mode; for details, see your system administrator.)

After it is set, a file's type is preserved from one revision to the next. You can change or override Perforce's record of a file's type by opening the file with the `-t filetype` flag:

- `p4 add -t filetype filespec` adds the files as the type you specify.
- `p4 edit -t filetype filespec` opens the file for `edit`; when the files are submitted, the new filetype takes effect.
- `p4 reopen -t filetype filespec` changes the type of a file that's already open for `add` or `edit`.

File types are specified as `[basetype]+modifiers`, or through the use of file type keywords. File type modifiers can 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`. You can determine the type of an existing file by using `p4 opened` or `p4 files` on the file.

Partial filetypes are also acceptable. For example, to change an existing `text` file to `text+x`, use `p4 reopen -t +x myscript.pl`. Most partial filetype modifiers are added to the filetype, but the storage modifiers (`+C`, `+D`, and `+F`) replace the file's storage method. To remove a modifier, you must specify the full filetype.

A file's type determines whether *full file* or *delta* storage is used on the Perforce server. By default, `binary` files are stored in full on the server. For `text` files, only the changes (the "deltas') associated with each revision are stored: this is called *delta storage*. Perforce uses RCS format to store deltas. When delta storage is used, you can perform merges and line-by-line file comparisons between file revisions. Files that are stored in their full form cannot be merged or compared.

Some file types are automatically compressed to `gzip` format when stored in the depot. The compression occurs when you submit the file, and decompression happens when you sync (copy the file from the server to the workspace). The client workspace always contains the file as it was submitted to the depot.

Warning! Do not try to use delta storage against binary files by manually changing the file type to `text`. Unpredictable results may occur if you attempt to submit a binary file with its filetype manually set to `text`.

Base file types

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

File type modifiers

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.

File type keywords

The following table lists the Perforce file type keywords and their equivalent base file types and modifiers:

Old Keyword	Description	Base Filetype	Modifiers
`text`	Text file	`text`	none
`xtext`	Executable text file	`text`	`+x`
`ktext`	Text file with RCS keyword expansion	`text`	`+k`
`kxtext`	Executable text file with RCS keyword expansion	`text`	`+kx`
`binary`	Non-text file	`binary`	none
`xbinary`	Executable binary file	`binary`	`+x`
`ctext`	Compressed text file	`text`	`+C`
`cxtext`	Compressed executable text file	`text`	`+Cx`
`symlink`	Symbolic link	`symlink`	none
`resource`	Macintosh resource fork	`resource`	none
`uresource`	Uncompressed Macintosh resource fork	`resource`	`+F`
`ltext`	Long text file	`text`	`+F`
`xltext`	Executable long text file	`text`	`+Fx`
`ubinary`	Uncompressed binary file	`binary`	`+F`
`uxbinary`	Uncompressed executable binary file	`binary`	`+Fx`
`tempobj`	Temporary object	`ubinary`	`+FSw`
`ctempobj`	Temporary object (compressed)	`cbinary`	`+Sw`
`xtempobj`	Temporary executable object	`ubinary`	`+FSwx`
`xunicode`	Executable unicode	`unicode`	`+x`

Overriding file types with the typemap table

Some file formats (for example, Adobe PDF files and Rich Text Format files) are actually `binary` files, but can sometimes be erroneously detected by Perforce as being of type `text`. Your system administrator can use the `p4 typemap` command to set up a table that matches file names to specific Perforce file types.

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 attempting to guess the file's type by examining its contents. To override file types in the typemap table, specify the file type on the command line with the `-t` *`filetype`* flag.

Preserving timestamps with the +m modifier

The default behavior of Perforce is to update the timestamp on files in your client workspace when you sync the files. If you need to preserve a file's original timestamp, use the modtime (`+m`) file type modifier. Doing so enables you to ensure that the timestamp of a file in a client workspace after a `p4 sync` is the original timestamp existing *on the file* at the time of changelist submission (that is, *not* the time at the Perforce server at time of submission, and *not* the time on the client workstation at the time of sync).

The `+m` modifier is useful when developing using 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. If you use the `+m` modifier on a file, Perforce ignores the `modtime` ("file's timestamp at time of submission") or `nomodtime` ("date and time on the client at time of sync") options in the `p4 client` form when syncing the file, and always restores the file's original timestamp at the time of changelist submission.

Expanding RCS keywords with the +k modifier

If you use the `+k` modifier to activate RCS keyword expansion for a file, 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 $`

Keyword	Expands To	Example
`$Revision$`	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`, display a form that you must fill in while using a text editor. When you change the form and exit the editor, the form is parsed by Perforce, checked for errors, and used to complete the command. If there are errors in the form, Perforce displays an error message and you must edit the form again.

The rules of form syntax are as follows: field names must be against the left margin and end with a colon, and field values must either be on the same line as the field name, or indented on the lines beneath the field name. 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 multiple values, one per line.

Certain fields, like `Client:` in `p4 client`, are read-only and cannot have their values changed; other fields, like `Description:` in `p4 submit`, must have their values changed. When in doubt about which fields can (or must) be modified, see the *Command Reference* or use `p4 help` *`command`*.

Changing the default forms editor

To override the default editor on either Windows or UNIX, set the Perforce environment variable `P4EDITOR` to the full path of your editor of choice.

On Windows, the default text editor for Perforce forms is Notepad. On UNIX, the default text editor for Perforce forms is whatever editor is specified by the `EDITOR` environment variable, or `vi` if no `EDITOR` is specified.

Scripting with Perforce forms

Any commands that require you to fill in a form, such as `p4 client` and `p4 submit`, can read a from standard input with the `-i` flag. Similarly, you can use the `-o` flag to direct a form 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's fields within your script, and then use the `-i` flag to send the processed form back to the Perforce client program.

For instance, to create a new job by means of a script, use `p4 job -o > tempfile` to write a blank job specification to a temporary file, then add information to the proper lines in *tempfile*, and then use a command such as `p4 job -i < tempfile` to read the edited form and store the job data in Perforce, just as if a user had entered the job data from within an editor.

The commands that display forms and support the `-i` and `-o` flags are:

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

Note `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.

The following Perforce reporting commands generate information on depot files, not files within the client workspace. When files are specified in local or client syntax on the command line, Perforce uses the client workspace view to map the specified files to their locations in the depot.

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 workspace *`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.

Consider the following scenario:

1. Ed opens `file.c` for edit in his client workspace.
2. Lisa opens the same file for edit in her client workspace.
3. Ed and Lisa both work on their respective versions of `file.c`.
4. Ed submits a changelist containing his version of `file.c`, and the submit succeeds.
5. Lisa submits a changelist with her version of `file.c`. Her submit fails because of a file conflict with Ed's version.

If the Perforce server were to accept Lisa's version into the depot, the head revision would contain none of Ed's changes. Therefore, Lisa's changelist is rejected and she must resolve the conflict. To resolve the conflict, she can:

- Submit her version of the file, overriding Ed's version.
- Discard her changes to the file in favor of Ed's version.
- Generate a *merged* version of the file that contains both sets of changes, and submit the merged version.
- Review and edit the merged version of the file before submitting it.

Resolving a file conflict is a two-step process: first, the resolve is *scheduled*, and after scheduling the resolve, the user who submitted the changelist that scheduled the resolve must perform the resolve.

The Perforce server automatically schedules a resolve whenever you submit a changelist fails due to a file conflict. You can also schedule a resolve manually by syncing the head revision of a file over an opened revision within your client workspace. To perform a resolve, use the `p4 resolve` command. Perforce also provides facilities to prevent file conflicts by locking files when they are edited.

Scheduling Resolves of Conflicting Files

Whenever you try to submit a file to the depot that is not an edit of the file's current head revision, a file conflict exists, and you must resolve the conflict.

The file revision that was most recently synced to your client workspace is the *base file revision*. If the base file revision for a particular file in your workspace is not the same as the head revision of the same file in the depot, you must perform a *resolve* the new file revision can be accepted into the depot.

Before you can perform a resolve with the `p4 resolve` command, you must schedule the resolve. You can schedule a resolve with `p4 sync`, or by submitting a changelist that contains the newly conflicting files. If a resolve is necessary, your submit fails, and the resolve is automatically scheduled for you.

Why "p4 sync" to schedule a resolve?

When you use `p4 sync`, you are projecting the state of the depot onto your client workspace. For each file on which `p4 sync` operates:

- If the file does not exist in your workspace, or it is found in your workspace but is unopened, the file is copied from the depot to your workspace.
- If the file has been deleted from the depot, it is deleted from your workspace.
- If you have opened the file in your workspace with `p4 edit`, the Perforce server cannot copy the file into your workspace, because doing so would overwrite any changes you had made. The Perforce server schedules a *resolve* between the file revision in the depot, the revision in your workspace, and the base file revision (that is, the revision you last synced to your workspace).

Example: *Scheduling resolves with* `p4 sync`

Ed is making a series of changes to the `*.guide` *files in the elm doc subdirectory. He has synced the* `//depot/elm_proj/doc/*.guide` *files to his workspace 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`*. Because these files are already open in his workspace, his workspace files are not overwritten. The* `p4` *client program schedules the resolves between Ed's workspace 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 have caused the* `p4 submit` *to fail, and the resolves would have been scheduled as a result of the submission failure.*

How do I know when a resolve is needed?

The `p4 submit` command fails when it determines that any of the files in the submitted changelist need to be resolved, and displays a message that includes the names of the files that you must resolve. If the changelist provided to `p4 submit` was the default changelist, the changelist is assigned a number that you must use in all future references to the changelist. Numbered changelists are discussed in Chapter 7, Changelists.

Another way to determine if a resolve is needed is to run `p4 sync -n` *`filename`* before performing the submit, using the open files in your changelist as the arguments to the command. If resolves on files are necessary, `p4 sync -n` reports them. If you use this approach, the files in your default changelist remain in your default changelist (that is, you do not need to create a numbered changelist).

Performing Resolves of Conflicting Files

You resolve file conflicts with `p4 resolve [`*`filenames`*`]`. Provide files to be resolved as arguments to the `p4 resolve` command. Each file is resolved separately.

The `p4 resolve` process starts with three revisions of the same file and generates a fourth version; you can accept any of these versions in place of the file in your workspace, and you can edit the generated version before accepting it into your workspace. Whatever version of the file you choose, after you have resolved the conflict, use `p4 submit` to submit the file to the depot.

The `p4 resolve` command is interactive, and presents a series of prompts:

```
/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 (*`yours`*, *`theirs`*, and *`base`*), generates a new version (*`merged`*) that merges elements of all three revisions, provides you with an opportunity to edit the merged file, and writes the resulting file (either *`yours`*, *`theirs`*, *`merged`*, or your edited *`merged`* file) to your client workspace.

The file revisions used by `p4 resolve` are as follows:

`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. After the differences are located, 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 do not need to be resolved. The number of line sets that apply to 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 file#n
(text from the original version)
==== THEIR VERSION file#m
(text from their file)
==== YOUR VERSION file
(text from your file)
<<<<
```

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. You can speed this process 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 conflict between the *yours* and *theirs* files. When resolving conflicts, remember that the absence of conflicting diff chunks does not imply correctness of code.

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
`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*

Option	Short Meaning	What it Does
`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 any changes 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 algorithm built into the `p4` client program. To use a third-party diff utility in place of the `p4` client program's diff algorithm, specify the third-party utility in the `P4DIFF` environment variable.

The `p4 resolve` prompt has the following format:

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

A recommended choice is displayed in brackets at the end of the prompt. Pressing `Return` or choosing `Accept` accepts the default option. The recommended option is chosen according 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 into your workspace, your previous workspace file is overwritten, and you must still submit the revised file to the depot. 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 when you call `p4 submit`; if this has happened, you will have to perform another resolve. (You can prevent this from happening by performing a `p4 lock` on the file before starting the resolve. See "Preventing multiple resolves with p4 lock" on page 76 for details.)

Command line flags to automate the resolve process

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.
`-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*; Changes that are common to *theirs* and *yours* are not considered to belong to either file, and are accepted; 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. In these circumstances, `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

After you have opened a file, you can lock it with `p4 lock` so that only you can submit the next revision of that file to the depot. After you submit a locked file, it is automatically unlocked. You can also manually unlock files you have locked by using the `p4 unlock` command.

The benefit of `p4 lock` is that after a file is locked, the user who locked it experiences no further conflicts on that file, and never needs to resolve the file. This benefit comes at a price: other users can open the file for edit, but are unable to submit changes affecting 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 supports a `+l` file type modifier to support exclusive-open (pessimistic locking). If you have a `+l` file open for edit, other users who attempt to edit the file receive an error message.

The difference between `p4 lock` and `+l` is that `p4 lock` permits anyone to open a file for edit, but only the person who locked the file can submit a changelist containing that file to the depot. By contrast, files of type `+l` can be opened for edit by only one user at a time.

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 `file` for edit.
2. Lisa opens the same file in her client for edit.
3. Ed and Lisa both edit their client workspace versions of `file`.
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 `file`.
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 can 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 using the `+l` modifier prevents concurrent development, for some file types (usually binary files), merging and resolving might not be meaningful, and some sites require pessimistic locking as a matter of policy.

Your Perforce administrator can use the `p4 typemap` command (see the *Perforce Command Reference*) to ensure that *all* files matching a file specification (or even all files sitewide) are assigned type `+l` by default.

Resolves and Branching

You also use `p4 resolve` to integrate changes between branches; 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. Use the `p4 help` command for a complete listing of the flags and options available with these reporting commands.

Chapter 6

Perforce Basics: Miscellaneous Topics

So far, this manual has provided an introduction to the basic functionality provided by Perforce. Subsequent chapters cover the more advanced features. Included here is information on the following miscellaneous topics likely to be encountered while working with Perforce:

- Using passwords and logging in and out of Perforce,
- Controlling your Perforce environment with the `P4CONFIG` environment variable,
- Command-line flags common to all `p4` commands,
- How to work on files while not connected to a Perforce server,
- Renaming files, and
- Recommendations for organizing files within the depot.

Perforce Passwords

Without passwords, any user can assume the identity of any other Perforce user by setting the value of `P4USER` to a different username, either by using the `-u` flag with the `p4` command, or by setting `P4USER` in a `P4CONFIG` configuration file. To improve security and accountability, use passwords.

Setting passwords

To prevent someone else from impersonating you within Perforce, set a password with the `p4 passwd` command. No one, including the user who set the password, will be able to use any Perforce commands under your username without providing the password to Perforce.

Your system administrator may have configured your Perforce server to require “strong” passwords. A password is considered strong if it is at least eight characters long, and at least two of the following are true:

- Password contains uppercase letters
- Password contains lowercase letters
- Password contains non-alphabetic characters.

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

Be careful when setting passwords. If you forget your password after having set it, a Perforce superuser will have to reset or remove your password for you.

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

Perforce passwords and authentication

To authenticate to a Perforce server with a password, set your password with `p4 passwd`, and use the password with every Perforce command you run. To use the password you have set, either:

- set the value of the environment or registry variable `P4PASSWD` to your password, or
- set the value of `P4PASSWD` within the file described by `P4CONFIG`, or
- use the `-P` *`password`* flag between `p4` and the actual command when calling Perforce commands (for instance, `p4 -P` *`mypassword`* `submit`).

If you are using ticket-based authentication, changing your password automatically logs out all of your existing tickets.

Perforce tickets: logging in and out

In addition to password-based authentication, Perforce supports ticket-based authentication. Because ticket-based authentication does not rely on environment variables or command-line flags, it is more secure than password-based authentication, and your system administrator may have configured your server to require its use.

By default, tickets are stored in a file in your home directory. To specify a different ticket file, set the `P4TICKETS` environment variable to the location of your ticket file.

After you have logged in, your ticket is valid for a limited period of time (by default, 12 hours). Your administrator may have changed this setting. By default, your ticket is valid only for the IP address of the workstation from which you logged in, but you can create tickets that are valid for any IP address if you are working from more than one machine and using a shared ticket file.

To use ticket-based authentication, set a password with `p4 passwd`, and log in by getting a ticket from the Perforce server by using the `p4 login` command.

```
p4 login
```

Enter your password at the prompt. If you log in successfully, a ticket is created for you in your ticket file, and you are not prompted to log in again until either your ticket expires (by default, in 12 hours), or until you log out of Perforce.

To extend a ticket's lifespan, run `p4 login` while already logged in. Running `p4 login` while already logged in extends your ticket's lifespan by 1/3 of its initial timeout period, subject to a maximum of its initial timeout period.

If you are in an environment where you are using Perforce from more than one machine while sharing the same ticket file (for example, many UNIX environments with shared home directories), log in with:

```
p4 login -a
```

Using `p4 login -a` creates a ticket in your ticket file that is valid from all IP addresses, enabling you to remain logged in to Perforce from more than one workstation.

To see if your ticket is still valid, use:

```
p4 login -s
```

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

To log out of Perforce by deleting your ticket, use:

```
p4 logout
```

If you are logged in from more than one machine at once, you can log out from all machines simultaneously by invalidating your ticket with:

```
p4 logout -a
```

For more information, see the *Command Reference*.

Reconfiguring the Perforce Environment with $P4CONFIG

Some Perforce users have multiple client workspaces, each of which can connect to a different Perforce server. There are three ways you can change 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` specifies 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`, `P4TICKETS`, 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	P4HOST
P4PORT	P4MERGE	P4PASSWD	P4TICKETS	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 you may have set. Command-line flags (discussed in the next section) override the values found in the `P4CONFIG` file.

The `P4CONFIG` variable is particularly useful if you use multiple client workspaces on the same workstation, because it automates the process of changing the Perforce environment variables whenever you switch your current working directory from one client workspace directory to another.

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`. The flags are as follows:

Flag	Meaning	Example
`-c` *`clientname`*	Runs the command on the specified client. Overrides the `P4CLIENT` environment variable.	`p4 -c joe edit //depot/file.c` Opens file `file.c` for editing under client workspace `joe`.
`-C` *`charset`*	For servers in unicode mode, override the `P4CHARSET` variable.	`p4 -C utf8 print //depot/file`
`-d` *`directory`*	Specifies the current directory, overriding the environment variable `PWD`.	`p4 -d ~elm/src edit one two` Opens files `one` and `two` for edit; these files are found relative to `~elm/src`.
`-G`	Cause all output (and batch input for form commands using the `-i` option) to be formatted as marshalled Python dictionary objects	`p4 -G info`
`-H` *`host`*	Specify the host name, overriding the environment variable `P4HOST`.	`p4 -H` *`host`* `print //depot/file`
`-L` *`language`*	For servers with non-English error messages, override the `P4LANGUAGE` variable.	Reserved for system integrators.
`-p` *`server`*	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`.
`-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.

Flag	Meaning	Example
`-s`	Prepend a tag to each line of output so as to make output more amenable to scripting.	`p4 -s info`
`-u` *`username`*	Specifies a Perforce user, overriding the `P4USER` environment variable. The user can 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 works 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 "Working Detached" on page 84.
`-V`	Displays the version of the `p4` executable.	`p4 -V`

All Perforce commands can take these flags, even commands for which these flag usages are useless (for instance, `p4 -u bill -d /usr/joe help`). Other flags are available as well; these additional flags are command dependent. See 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 making the file writable in your client workspace.

If you expect not to have network connectivity to the server for a prolonged period of time, you will need to work detached from the server. To work detached, use the following techniques:

1. Work on files without using Perforce commands. Instead, use native OS commands to manually change the permissions on files, and then edit or delete the files.
2. If you did not edit the files within a client workspace, copy the files to your client workspace when the network connection is reestablished.
3. Use `p4 diff -se` and `p4 diff -sd` to find files in your workspace that have changed without Perforce's knowledge (that is, without the use of Perforce commands). Use the information provided by the `p4 diff -se` and `p4 diff -sd` commands to bring the depot in sync with the client workspace.

Finding changed files

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.

Updating the depot with changed files

You can use the -x flag in combination with the output of the `p4 diff -se` and `-sd` commands to bring the state of the depot in sync with the changes you made to your workspace while disconnected.

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 you removed from your 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

If you accidentally use the local OS file deletion or permission modification command, the Perforce server can lose track of the state of your client workspace. For example, suppose you accidentally delete a file in your workspace with the UNIX `rm` command. Even after you submit the changelist, `p4 have` still lists 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 Perforce server *thinks* you have. Using `p4 sync -f` is mostly a recovery tool for bringing the client workspace back into sync with the depot after accidentally removing or changing files managed by Perforce.

Renaming Files

To rename files, use `p4 integrate` to copy the file from one location in the depot to another, delete the file from the original location, and then submit 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/one` and *`to_file`* is `d2/two`, then `one` is moved to the `d2` directory, and is renamed `two`. The *`from_file`* and *`to_file`* specifiers can include wildcards, as long as they are matched on both sides.

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 the file 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.

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 workspace 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 `zeus`, `athena`, and `apollo`, three subtrees might be created within the depot: `//depot/zeus`, `//depot/athena`, and `//depot/apollo`. If Joe is working on the `zeus` project, his mapping might look like this:

```
//depot/zeus/...    //joe/...
```

And Sarah, who’s working on the `athena` and `apollo` projects, might set up her client workspace as:

```
//depot/athena/...    //sarah/athena/...
//depot/apollo/...   //sarah/apollo/...
```

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

Another way of solving the same problem is to have the Perforce system administrator create one depot for each project or branch. See the *Perforce System Administrator’s Guide* for details about setting up multiple depots.

Chapter 7 Changelists

A Perforce *changelist* is a list of files, their revision numbers, and operations to be performed on these files. You add files to a changelist with commands such as `p4 add` *`filenames`* or `p4 edit` *`filenames`*, and the changed files are stored in the depot when you submit the changelist with `p4 submit`.

When you submit a changelist 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 all files grouped together in a changelist are updated simultaneously. A Perforce changelist is an *atomic change transaction*.

Perforce attempts to make working with changelists as transparent as possible. 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. Sometimes, the default changelist is not sufficient, and you must use a *numbered changelist*. Use numbered changelist when:

- You are working on two or more unrelated changes at the same time.

  For example, suppose you're fixing two bugs, each of which affects a separate set of files. Rather than submit the fixes to both bugs in a single changelist, you might create one changelist for the files that fix the first bug, and a second changelist for the files that fix the second bug, and use two `p4 submit` commands to submit your work to the depot.

- The `p4 submit` command fails.

  For example, if you are working on a file in the default changelist, but another user has locked the file, or submitted a changelist that affects your file, your submit fails. Whenever 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 assigned number.

Working with the Default Changelist

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 file in a changelist is said to be *open* in the client workspace and in the changelist: the first file in the example is open for edit, and the second file is open for deletion.

The depot is updated with the files in the changelist when you call `p4 submit` to send your changes to the Perforce server.

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`, operate 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`, a change form is displayed, showing the files in the default changelist. In order to submit a changelist, you must also supply a description of the changes being made. When you save the `p4 submit` form, the files shown in the form are submitted to the server and the server attempts to update the files in the depot.

If there are no problems with the submission, the changelist is assigned a sequential number, and its status changes from `new` or `pending` to `submitted`. Any files you removed from the default changelist by editing the `p4 submit` form reappear in a new default changelist. After 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

You can create a numbered changelist manually by using the `p4 change` command. This command brings up the same form that you see during `p4 submit`.

All files in the default changelist are included in the new changelist. When you exit the form, the new changelist is assigned the next changelist number in sequence, and the changelist must be subsequently referred to by this change number. You can delete files from the changelist by editing the `p4 change` form; files deleted from the new changelist reappear in the default changelist. The status for a changelist created by the `p4 change` command is `pending` until you submit the changelist.

Files in your workspace may appear in only one pending changelist at a time.

Working With Numbered Changelists

To add files to numbered changelists (as opposed to the default changelist), use the `-c` *`changenum`* flag when you use Perforce commands such as `p4 edit` *`filename`*. For example, if you have just created a changelist 34, use `p4 edit -c 34` *`filename`* to add *`filename`* to pending changelist 34.

You can move files from one changelist to another with `p4 reopen -c` *`changenum filename`*, where *`changenum`* is the number of the moving-to changelist. To move a file from a numbered changelist to the default changelist, use `p4 reopen -c default` *`filename`*.

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 fix each bug in its own changelist; doing so enables 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, and 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 because the affected files are in the default changelist, he submits the default changelist with `p4 submit` *and no arguments. When Ed finishes fixing the filtering bug, he submits that changelist with* `p4 submit -c 29`.

Automatic Creation and Renumbering of Changelists

Changelist submission can fail for a number of reasons:

- A file in the changelist has been locked by another user with `p4 lock`.
- Your client workspace no longer contains a file included in the changelist.
- There is a server error, such as not enough disk space.
- You were not editing the head revision of a particular file. (If another user submits a change to a file you're working on, the revision in your client workspace is no longer the head revision.)

If any of the files 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, the Perforce server assigns the next available changelist number to the changelist, and this change number must be used to refer to the changelist in the future. Perforce also locks the files to prevent others from changing them while you resolve the reason for the failed submit.

If a submit fails because your revision of the file is not the head revision, this is called a file *conflict*. You must *resolve* the conflict before the changelist can be accepted. (For details, see Chapter 5, Perforce Basics: Resolving File Conflicts).

When Perforce renumbers changelists

The changelist numbers of submitted changelists always reflect the order in which changelists were submitted to the depot. Whenever a numbered changelist is submitted out of sequence, the server automatically renumbers the changelist to reflect the order in which changelists were submitted.

Example: *Automatic renumbering of changelists*

Ed has finished fixing the filtering bug that he's been working on in numbered changelist 29. After Ed created that changelist, he submitted another changelist (change 30), and two other users have also submitted changelists. Ed submits change 29 with `p4 submit -c 29`*, and is informed of a changelist renumbering as follows:*

```
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`*. You must remove all files and jobs from a pending changelist before you can delete it. Use `p4 reopen` to move files to another changelist, `p4 revert` to remove files from the changelist (and to revert them back to their old versions), and `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`. Use `p4 changes` to obtain lists of changelists with short descriptions, and `p4 describe` to obtain verbose information pertaining to a specified 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

Labels provide a method of naming important sets of file revisions for future reference. You can use labels to reproduce the state of a client workspace, to branch files, and to compare files. For example, you might want to tag the file revisions that compose a particular release with the label `release2.0.1`. At a later time, you can retrieve all the tagged revisions into a client workspace by syncing the workspace to the label.

Use a label when you want to:

- keep track of all the file revisions contained in a particular release of software,
- distribute a particular set of file revisions to other users, or
- branch from a known set of file revisions.

Labels or changelist numbers?

Labels share certain important characteristics with changelist numbers, as both refer to particular sets of file revisions, and both can be used to refer to all the revisions in the set. Labels and changelists have some important distinctions:

- The files and revisions tagged by a label can be changed at any point in the label's existence.
- Changelists are always referred to by Perforce-assigned numbers, but labels are named by the user.
- A changelist is an implicit revision marker, but labels must be applied in order to be used as revisions.
- A changelist is a revision marker that applies to all files in the depot, but labels are typically limited to a subset of depot files.

Using labels

Note Versions of Perforce prior to release 2004.2 used the `p4 labelsync` command to tag files with a label. The `p4 tag` command, introduced in Release 2004.2, simplifies this process. The `p4 labelsync` command remains available for specialized purposes.

For details, see "Differences between p4 tag and p4 labelsync" on page 100.

Tagging files with a label

To tag a set of file revisions (in addition to any revisions that have already been tagged), use `p4 tag`, specifying a label name and the desired file revisions. New label names must be distinct from any existing client workspace, branch, or depot names.

For example, to tag the head revisions of files that reside under `//depot/proj/rel1.0/hdrs` with the label `my_label`, use the following:

```
p4 tag -l my_label //depot/proj/rel1.0/hdrs/...
```

The head revisions of files under `//depot/proj/rel1.0/hdrs/...` are tagged with the name `my_label`.

To tag revisions other than the head revision, specify a changelist number in the file pattern:

```
p4 tag -l my_label //depot/proj/rel1.0/hdrs/...@1234
```

Untagging files

You can untag revisions with:

```
p4 tag -d -l labelname filepattern
```

This command removes the association between *labelname* and all files tagged by *labelname*, regardless of the revision specified in the *filepattern*.

To untag a subset of tagged files, supply a file specification. For example, if you have previously tagged all revisions under `//depot/proj/rel1.0/...` with `my_label`, you can untag only the header files with:

```
p4 tag -d -l my_label //depot/proj/rel1.0/hdrs/*.h
```

Revisions of the `*.h` header files are no longer tagged with `my_label`.

Previewing tag's results

You can preview the results of `p4 tag` with `p4 tag -n`. This command lists the revisions that would be tagged, untagged, or retagged by the `tag` command without actually performing the operation.

Listing files tagged by a label

To list the revisions tagged with *labelname*, use `p4 files`, specifying the label name as follows:

```
p4 files @labelname
```

All revisions tagged with *labelname* are listed with their file type, change action, and changelist number. (This command is equivalent to `p4 files //...@labelname`)

Listing labels that have been applied to files

To list the labels that have been applied to any of the files in a filepattern, use the command:

```
p4 labels filepattern
```

Referring to files using a label

You can use a label name anywhere you can refer to files by revision (`#1`, `#head`), changelist number (`@7381`), or date (`@2003/07/01`).

If `p4 sync @labelname` is called with no file parameters, all files in the workspace view that are tagged by the label are synced to the revision specified in the label. All files in the workspace that do not have revisions tagged by the label are deleted from the workspace. Open files or files not under Perforce control are unaffected. This command is equivalent to `p4 sync //...@labelname`.

If `p4 sync @labelname` is called with file arguments, as in `p4 sync files@labelname`, files in the user's workspace that are specified on the command line and also tagged by the label are updated to the tagged revisions.

Example: *Retrieving files tagged by a label into a client workspace.*

Lisa wants to retrieve some of the binaries tagged by Ed's `build1.0` *label into her client workspace. To get all files tagged by* `build1.0`*, she could type:*

```
p4 sync //depot/...@build1.0
```

or even:

```
p4 sync @build1.0
```

Instead, she's interested in seeing only one platform's build from that label, so she types:

```
p4 sync //depot/proj/rel1.0/bin/osx/*@build1.0
```

and sees:

```
//depot/proj/rel1.0/bin/osx/server#6 - added as /usr/lisa/osx/server#6
//depot/proj/rel1.0/bin/osx/logger#12 - added as /usr/lisa/osx/logger#12
//depot/proj/rel1.0/bin/osx/install#2 - added as /usr/lisa/osx/install#2
<etc>
```

All files under `//depot/proj/rel1.0/bin/osx` *that are tagged with Ed's* `build1.0` *label and are also in Lisa's client workspace view are retrieved into her workspace.*

Deleting labels

To delete a label, use the following command:

```
p4 label -d labelname
```

Deleting a label has no effect on the tagged files other than to remove the ability to refer to the files with the `@labelname` revision specifier.

Creating a label for future use

The `p4 tag` command both creates a label and applies it to files. To create a label without tagging any file revisions, use `p4 label labelname`. This command brings up a form similar to the `p4 client` form. After you have created a label, you can use `p4 tag` or `p4 labelsync` to apply the label to file revisions.

Label names share the same namespace as client workspaces, branches, and depots. A new label name must be distinct from any existing client workspace, branch, or depot name.

For example, you can create a new label `my_label` by typing:

```
p4 label my_label
```

The following form is displayed:

```
Label: my_label
Owner:  edk
Description:
        Created by edk.
Options:        unlocked
View:
        //depot/...
```

Enter description for the label, and save the form. (You do not need to change the `View:` field.)

After creating the label, you are able use the `p4 tag` and `p4 labelsync` commands to apply the label to file revisions. Only one revision of a given file can be tagged with a given label, but the same file revision can be tagged by multiple labels.

Using label views

The `View:` field in the `p4 label` form limits the files that can be tagged with a label. To include files and directories in a label view, specify the files and directories to be included using depot syntax.

To prevent yourself from inadvertently tagging your entire workspace with a label, set the label's `View:` field to a subset of the depot. The default label view includes the entire depot (`//depot/...`), which tags any (and every) file in the depot with `p4 labelsync`.

Example: *Using a label view to control what files can be tagged*

Ed wishes to tag a recently built set of binaries with the label `build1.0`*. He wants to ensure that only the files in the build path can be tagged with the* `build1.0` *label.*

He types `p4 label build1.0` *and uses the label's* `View:` *field to restrict the scope of the label as follows:*

```
Label: build1.0
Owner:  edk
Description:
        Created by edk.
Options:        unlocked
View:
        //depot/proj/rel1.0/bin/...
```

After he saves from the editor, a new label `build1.0` *is created. This label can tag only files in the* `/rel1.0/bin` *directory.*

With the default `View:` *of* `//depot/...`, *Ed must type:*

```
p4 tag -l build1.0 //depot/proj/rel1.0/bin/...
```

With the new label view, he can use the simpler `p4 tag -l build1.0 //...` *to tag the desired files.*

Using labels to record workspace configurations

The `p4 labelsync` command is a variant of `p4 tag` that you can use to record a workspace configuration. For example, to record the configuration of your current client workspace using the pre-existing `ws_config` label, use the following command:

```
p4 labelsync -l ws_config
```

All file revisions synced to your current workspace are tagged with the `ws_config` label. Files previously tagged with `ws_config` are untagged unless they are also synced to your workspace.

To recreate the workspace tagged by the `ws_config` label, sync your workspace to the label with:

```
p4 sync @ws_config
```

Preventing inadvertent tagging and untagging of files

Using `p4 labelsync` with no file arguments tags the eligible files in your client workspace and any applicable label view, and untags all other files. This means that it is possible to accidentally lose the information that a label is meant to preserve.

To prevent the inadvertent tagging and untagging of files, lock the label by setting the value of the `Options:` field of the `p4 label` *`labelname`* form to `locked`. Other users will be unable to use `p4 labelsync` or `p4 tag` to tag files with that label until you or a Perforce superuser unlocks the label.

Differences between p4 tag and p4 labelsync

The differences between `p4 tag` and `p4 labelsync` are as follows:

- The `p4 tag` command automatically creates and applies a new label to specified files if no label exists. The `p4 labelsync` command can only be used to tag files with an existing label.
- The `p4 tag` command requires a file pattern argument. The `p4 labelsync` command does not require a file pattern argument; if you call `p4 labelsync` without a file pattern argument, the command applies to your entire client workspace view.
- By default, `p4 tag` tags the `#head` revisions of files in the depot. By default, `p4 labelsync` tags the revisions of files most recently synced to your client workspace; that is, to the `#have` revision.
- A single call to `p4 tag` can tag or untag files, but never both. The `p4 labelsync` command can tag some files while untagging others in order to make a label match a client workspace.

How p4 tag works

The full syntax of the `p4 tag` command is:

```
p4 tag [-d -n] -l labelname filename...
```

The rules followed by `p4 tag` to tag files with a label are as follows:

1. If *`labelname`* does not exist, create *`labelname`*. By default, you are the owner of the new label, the label is `unlocked`, and has a view of the entire depot. After creating the label, you can change these default options by using `p4 label` *`labelname`*.
2. If *`labelname`* already exists, you must be the owner of the label to use `p4 tag` on it, and the label must be `unlocked`. If you are not the owner of a label, you can (assuming you have sufficient permissions) make yourself the owner by running:

   ```
   p4 label labelname
   ```

   and changing the `Owner:` field to your Perforce user name in the `p4 label` form. If you are the owner of a label, you can unlock the label by setting the `Options:` field (also in the `p4 label` form) to `unlocked`.
3. All files tagged with a pre-existing label must be in the label view. Any files or directories not included in a label view are ignored by `p4 tag`.
4. When `p4 tag` *`files`* is used to tag specified file revisions with a label, revisions not tagged with the label are tagged. If another revision of the file has already been tagged with the label, it is untagged and the newly specified revision is tagged. Any given file revision can be tagged by one or more labels, but only one revision of any file can be tagged with a given label at any one time.

5. If you call `p4 tag` with file pattern arguments that contain no revision specifications, the `#head` revisions are tagged with the label.
6. If you call `p4 tag` with file pattern arguments and those arguments contain revision specifications, the specified file revisions are tagged with the label.

How p4 labelsync works

The full syntax of the `p4 labelsync` command is:

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

The rules followed by `labelsync` to tag files with a label are as follows:

1. A label must exist, and you must be the owner of the label to use `p4 labelsync` on it, and the label must be `unlocked`.
2. All files tagged with a label must be in the label view specified in the `p4 label` form. Any files or directories not included in a label view are ignored by `p4 labelsync`.
3. When `p4 labelsync` is used to tag a file revision with a label, the revision is tagged with the label if it is not already tagged with the label. If a different revision of the file is already tagged with the label, it is untagged and the newly specified revision is tagged. Any given file revision can be tagged by one or more labels, but only one revision of any file can be tagged with a given label at any one time.
4. If `labelsync` is called with no filename arguments, as in:

   ```
   p4 labelsync -l labelname
   ```

   then all the files in both the label view and the client workspace view are tagged with the label. The revisions tagged by the label are those last synced into the client workspace; these revisions ("the `#have` revisions") can be seen in the `p4 have` list. Calling `p4 labelsync` this way removes the label from revisions it previously tagged unless those revisions are in your workspace.
5. When you call `p4 labelsync` with file pattern arguments, but the arguments contain no revision specifications, the `#have` revision is tagged with the label.
6. If you call `p4 labelsync` with file pattern arguments and those arguments contain revision specifications, the specified file revisions are tagged with the label.

   Specifying a revision in this manner overrides all other ways of specifying a revision with a label; whether the client workspace contains a different revision than the one specified, (or doesn't contain the file at all), the revision specified on the command line is tagged with the label.

The following table lists variations of `p4 labelsync` as typed on the command line, their implicit arguments as parsed by the Perforce Server, and the sets of files from which `p4 labelsync` selects the intersection for tagging.

Call p4 labelsync with	Implicit Arguments
`-l label` (no files specified)	`-l label //`*`myworkspace`*`/...#have` Tags every file in your client workspace at the revision currently in your client workspace.
`-l label `*`files`* (files specified in local syntax, no revision specified)	`-l label [`*`cwd`*`]/`*`files`*`#have` Tags only the files in your client workspace that you specify, at the revision currently in your client workspace.
`-l label `*`files#rev`* (files specified in local syntax, specific revision requested)	`-l label [`*`cwd`*`]/`*`files#rev`* Tags only the files in your client workspace that you specify, at the revision you specify. Files must be in your client workspace view. You can use numeric revision specifiers here, or `#none` to untag files, or `#head` to tag the latest revision of a file, even if you haven't synced that revision to your workspace.
`-l label //`*`files`* (files specified in depot syntax, no revision specified)	`-l label //`*`files`*`#have` Tags only the files in the depot that you specify, at the revision currently in your client workspace, whether the files are in your client workspace view or not.
`-l label //`*`files#rev`* (files specified in depot syntax, specific revision requested)	`-l label //`*`files#rev`* Tags only the files in the depot that you specify, at the revision at the revision you specify, whether the files are in your client workspace view or not.

Label Reporting

The commands that list information about labels are:

Command	Description
`p4 labels`	List the names, dates, and descriptions of all labels known to the server
`p4 labels `*`file#revrange`*	List the names, dates, and descriptions of all labels that tag the specified revision(s) of *`file`*.

Command	Description
`p4 files @`*`labelname`*	Lists all files and revisions tagged by *`labelname`*.
`p4 sync -n @`*`labelname`*	Lists the revisions tagged by the label that would be brought into your client workspace, (as well as files under Perforce control that would be deleted from your client workspace because they are *not* tagged by the label), without updating your client workspace.

Chapter 9 Branching

Perforce's *Inter-File Branching* mechanism enables you to copy any set of files within the depot and to track the changes between the copies. When you branch a set of files (or *codeline*) from one area of the depot to another, the new set of files evolves separately from the original files, but you can propagate changes in either codeline to the other with the `p4 integrate` command.

What is Branching?

Branching is a means of keeping two or more sets of similar (but not identical) files synchronized. Most software configuration management systems have some form of branching; Perforce's mechanism is particularly efficient 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 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.

At the time of release, branch a 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 Mac OS X driver using the UNIX code as a 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 `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 in the release branch and integrate changes intended to apply to all releases of the software 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/...` *must be branched into the* `//depot/elm_r2.0/...` *portion of the depot.*

Ed's client workspace root is `/usr/edk/elm_proj`*. Ed decides to work on the new* `//depot/elm_r2.0/...` *files in* `/usr/edk/elm_proj/r2.0`*. He uses the* `p4 client` *command to add the following mapping to his workspace view:*

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

He then runs:

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

to copy all the files under `//depot/elm_proj/...` *to* `//eds_elm/r2.0` *in his client workspace. Finally, he runs* `p4 submit` *to add the newly branched files to the depot.*

Why not just copy the files?

Although it is possible to accomplish everything that has been done so 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*, enabling you to easily and consistently track and propagate changes between one set of files and another.

Branching not only enables you to more easily track changes, it creates less overhead on the server. When you copy files with `p4 add`, you create two copies of the same file on the server. When you use branching, Perforce performs a "lazy copy" of the file, so that the depot holds only one copy of the original file and a record that a branch was created.

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 114).

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 "Command line flags to automate the resolve process" on page 74 for details.)

In most cases, there is *no* difference between branched files and non-branched files. Use the standard Perforce commands like `sync`, `edit`, `delete`, `submit`, and so on with all files, and permit both codelines to evolve separately. If changes to one codeline need to be propagated to another, use `p4 integrate` to propagate the changes. 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

You can propagate a change 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* `//depot/r2.0/...` *and creates copies of those files in his client workspace under* `//eds_elm/r2.0`*. He then runs* `p4 submit` *to add the newly branched files to the depot.*

After the branch has been created and the files have been copied into the branched codeline, you can propagate changes 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.
- Like client views, branch specifications can contain multiple mappings. 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 can 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`*. Because 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 enables you to integrate specific file revisions, to re-integrate and re-resolve previously-integrated code, and to merge two files that have no shared history.

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. The paper is available from:

```
http://www.cmcrossroads.com/bradapp/acme/branching/
```

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 on invalid input”, 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 enables you to link jobs 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 numbered 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 *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.

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
Status: open
User:   edk
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`.

Because job fields differ from site to site, the fields in jobs at your site might 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 permitted in job names.	`new`
`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.
`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.
`Date`	The date the job was last modified, displayed as *YYYY/MM/DD HH/MM/SS*	The date and time at the moment this job was last modified.

Field Name	Description	Default
`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.)

Using a custom job specification

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 might be additional fields in your `p4 job` form, and the names of the fields described above might have changed.

A sample customized job specification might look like this:

```
# Custom Job fields:
# Job:    The job name. 'new' generates a sequenced job number.
# Status: Either 'open', 'closed', or 'suspended'. Can be changed
# User:   The user who created the job. Can be changed.
# Date:   The date this specification was last modified.
# Type:          The type of the job. Acceptable values are
#                'bug', 'sir', 'problem' or 'unknown'
# Priority:      How soon should this job be fixed?
#                Values are 'a', 'b', 'c', or 'unknown'
# Subsystem:     One of server/gui/doc/mac/misc/unknown
# Owned_by:      Who's fixing the bug
# Description: Comments about the job. Required.

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

Some of the fields have been set by the administrator to use one value out 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 jobs -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 "`*`" enables you to perform 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 `&`). For instance, the command `p4 jobs -e '^user=edk'` is not permitted.

You can use the `*` wildcard to work around this restriction: `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, for example `Linda Hopper`
`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 three 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.
- manually, by editing them within the `p4 submit` form.

Linking jobs to changelists with the JobView: field

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 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`*.*

Linking jobs to changelists with p4 fix

`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.*

You can use `p4 fix` *to link a changelist to a job owned by another user.*

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 by typing:*

```
p4 fix -c 18 options-bug
```

Because changelist 18 has already been submitted, the job's status is changed to `closed`*.*

Linking jobs to changelists when submitting

You can also add jobs to changelists by editing the `Jobs:` field (or creating a `Jobs:` field if none exists) in the `p4 submit` form.

Any job can be linked to a changelist by adding it to a changelist's change form, or unlinked from a changelist by deleting the job from the 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: eds_ws
User:   edk
Status: new
Description:
        Updating "File" I/O 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
```

Because this job is unrelated to the work he's been doing, and since it hasn't been fixed, he deletes `job000125` *from the form and then quits from the editor. The changelist is submitted without* `job000125` *being associated with the changelist.*

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.

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

Ed uses the reporting commands to read the details about job `job000125`*. He fixes this problem, and a number of other bugs; when he next types* `p4 submit`*, he sees:*

```
Change: new
Client: eds_ws
User:   edk
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
```

Because 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`*.*

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 relationship between files, changelists, and their associated jobs.

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`*

To See a Listing of...	Use This Command:
...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 139, as well as the *Perforce Command Reference*.

Chapter 11 Reporting and Data Mining

Perforce's reporting commands supply information on all data stored within the depot. Many of these reporting commands have already been mentioned in 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 all the commands, see the *Perforce Command Reference*, or type `p4 help` *`command`* at the command line.

When you use file specifications with Perforce commands, *`filespec`* arguments such as:

```
p4 files filespec
```

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. See "Specifying File Revisions" on page 53 for more about revision specifiers.

Files

Commands that report on files fall into three categories: commands that provide information about file *contents*, (for instance, `p4 print`, `p4 diff`), commands that provide information about the state of the mapping between your depot and the client workspace (such as `p4 where` and `p4 have`), and commands that provide information on file *metadata*, the data that describe a file without regards to file content (such as `p4 files` and `p4 filelog`).

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. Because `p4 filelog` lists information about each revision of the specified file(s), file arguments to `p4 filelog` cannot contain revision specifications.

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 want to know where a particular client file is mapped to within the depot, or whether or not the head revision of a particular depot file has been copied to a client workspace.

The commands that reveal 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 of these commands can be used with or without *`filespec`* arguments. `p4 sync -n` is the only command in this set that permits the use of 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 or changelists affected lines in a text 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 displays all lines, including lines no longer present at the head revision, and associated revision ranges. The `-c` option displays changelist numbers, rather than revision numbers.

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 third line is deleted and the second line edited so that `file.txt#2` *reads:*

```
This is a text file.
The second line is new.
```

Finally, a third changelist is submitted, that includes no changes to `file.txt`*. After the third changelist, the output of* `p4 annotate` *and* `p4 annotate -c` *look like this:*

```
$ p4 annotate file.txt

//depot/files/file.txt#3 - edit change 153 (text)
1: This is a text file.
2: The second line is new.

$ p4 annotate -c file.txt

//depot/files/file.txt#3 - edit change 153 (text)
151: This is a text file.
152: The second line is new.
```

The first line of `file.txt` *has been present since* `file.txt#1`*, as submitted in changelist 151. The second line has been present since* `file.txt#2`*, as submitted in changelist 152.*

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 first line of output shows that the first line of the file has been present for revisions 1 through 3. The next two lines of output show lines of `file.txt` *present only in revision 1. The last line of output shows that the line added in revision 2 is still present in revision 3.*

You can combine the `-a` *and* `-c` *options to display all lines in the file and the changelist numbers (rather than the revision numbers) at which the lines existed.*

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, see 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, or even revisions of different files. The `p4 diff2` command takes two file arguments: wildcards are permitted, but any wildcards in the first file argument must be matched with a corresponding wildcard in the second. Using matching wildcards in `p4 diff2` 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, ignoring whitespace	`p4 diff -dw `*`file`*

The last example above bears further explanation; to understand how `diff -dw` works, it is necessary to discuss how Perforce implements and calls underlying diff routines.

Perforce uses two separate diff routines: one that is built into the `p4d` server, and another is used by the `p4` client. Both diffs contain identical, proprietary code, but are used by separate sets of commands. The client-side diff is used by `p4 diff` and `p4 resolve`, and the server-side diff is used by `p4 describe`, `p4 diff2`, and `p4 submit`.

The diff algorithm accepts multiple options, including a `-d` flag that enables you to generate RCS-style diffs, context diffs, unified diffs, and to determine what forms of whitespace (if any) are accounted for during file comparison.

The Perforce server always uses Perforce's diff algorithm, but Perforce client programs can also use third-party diff utilities. To use a third-party diff utility, set the `P4DIFF` environment variable to the full path name of the utility, and pass flags to the specified diff program with the `-d` flag, just as you pass flags to the built-in diff routine. Flags passed to the underlying diff are subject 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.

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 support the use of revision ranges with file specifications, see "Specifying ranges of revisions" on page 56.

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 139.

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 120)	`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 enable you to report only the set of changes that fix a particular job, the jobs fixed by a particular changelist, or jobs fixed by changelists that are linked to particular files.

A fixed job does not necessarily have a status of `closed`, because `open` jobs can be linked to pending changelists, and because `pending` jobs can be reopened even after the associated changelist has been submitted.

Other job reporting commands include:

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`*
...all jobs fixed by changelists that contain particular files	`p4 fixes` *`filespec`*

To See a Listing of...	Use This Command:
...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`*
...all jobs still open.	`p4 jobs -e status=open`

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`*

Listing Users, Workspaces, and Depots

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. See the *Perforce System Administrator's Guide* for more about using more than one depot on a single Perforce server.

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`

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 at least one Perforce client program (such as `p4` on UNIX, or `p4.exe` or `p4win.exe` on Windows).

The server and client executables 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.

UNIX Installation

Although you can install `p4` and `p4d` in any directory, on UNIX, the Perforce client programs typically reside in `/usr/local/bin`, and the Perforce server is usually located either in `/usr/local/bin` or in its own server root directory. 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, ensure that the `p4d` executable is owned and run by a Perforce user account that has been created for the purpose of running the Perforce server.

To start using Perforce:

1. Download the `p4` and `p4d` files for your platform from the Perforce web site.
2. Make the downloaded `p4` and `p4d` files executable.
3. Create a server root directory to hold the Perforce database and versioned files.
4. Tell the Perforce server what port to listen to by specifying a TCP/IP port to `p4d`.
5. Start the Perforce server (`p4d`).
6. Specify the name or TCP/IP address of the Perforce server machine and the `p4d` port number to the Perforce client program(s) by setting the `P4CLIENT` environment variable.

Download the files and make them executable

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

```
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*.

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

A Perforce server requires no privileged access; there is no need to run `p4d` as `root` or any other privileged user. See the *System Administrator's Guide* for details.

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

Telling the Perforce server which port to listen to

The Perforce server and client programs communicate with each other using TCP/IP. When `p4d` starts, it listens (by default) on port `1666`. Perforce client programs assume (also by default) that the `p4d` server is located on a host named `perforce`, listening on port `1666`.

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

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

Starting the Perforce server

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

```
p4d &
```

Although the example shown is sufficient to run `p4d`, other flags 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*.

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.

Stopping the Perforce server

To shut down a Perforce server, use the command:

```
p4 admin stop
```

to gracefully shut down the Perforce server. Only a Perforce superuser can use `p4 admin stop`.

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

Windows Installation

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

The Perforce installer allows you to:

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

  This option allows you to install `p4.exe` (the Perforce Command-Line Client), `p4win.exe` (P4Win, the Perforce Windows Client), and `p4scc.dll` (Perforce's implementation of the Microsoft common SCM interface).

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

  These options allow you to install Perforce client programs and the Perforce Windows server (`p4d.exe`) and service (`p4s.exe`) executables, or to automatically upgrade an existing Perforce server or service running under Windows.

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

- Uninstall Perforce: remove the Perforce server, service, and client executables, registry keys, and service entries. The Perforce database and the depot files stored under your server root are preserved.

Windows services and servers

The terms “Perforce server” and “`p4d`” are used interchangeably to refer to “the process which handles requests from Perforce client programs”. 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, the back-end program can be started either as a Windows service (`p4s.exe`) process that runs at boot time, or as a server (`p4d.exe`) process that must be invoked from a command prompt.

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

In most cases, it is preferable to install Perforce as a service, not a server. For a more detailed discussion of the distinction between services and servers, see the *Perforce System Administrator’s Guide.*

Starting and stopping Perforce

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

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

To stop a Perforce service (or server) at Release 99.2 or above, use the command:

```
p4 admin stop
```

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

For older revisions of Perforce, shut down services manually by using the **Services** applet in the **Control Panel**. Shut down servers running in command prompt windows by typing CTRL-C in the window or by clicking on the icon to Close the command prompt window.

Although these manual shutdown options work with 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, the manual shutdown options are obsolete.

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 use. 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
`P4ROOT`	Directory in which `p4d` stores its files and subdirectories
`P4TICKETS`	Specifies the location of the ticket file
`P4USER`	The user's Perforce username
`PWD`	The directory used to resolve relative filename arguments to `p4` commands
`TMP`	The directory to which Perforce writes its temporary files

Setting and viewing environment variables

Every operating system and shell has its own syntax for setting environment variables. The following table shows how to set the `P4CLIENT` environment variable in several operating systems and command shells.

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 learn more 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, `p4 set` displays the values of the associated environment variables. On Windows, `p4 set` displays either the MS-DOS environment variable (if set), or the value as set in the registry and whether the value was defined with `p4 set` (to apply to only the current user) or `p4 set -s` (to apply to all users on 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 managed by Perforce. By default, the client workspace name assumed to be name of the host machine on which the client workspace is located; set the `P4CLIENT` environment variable to override the default name, 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 Command-Line Client program, and the command you issue to execute Perforce commands from the operating system command line.
`p4d`	The Perforce Server; a program running on a Perforce server machine that manages the depot and the metadata for a Perforce installation.
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.
P4V	The Perforce Visual Client, a graphical interface to Perforce for Linux, Mac OS X, and Windows.
P4Web	The Perforce Web Client, a web-based interface to Perforce.

Term	Definition
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.
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.

Term	Definition
revision range	A range of revision numbers for a specified file, specified as the low and high end of the range. For example, `file#5,7` specifies revisions 5 through 7 of file `file`.
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.

Term	Definition
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.
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` is 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