Perforce 97.3 Command Line User's Manual

Manual 97.3m1

November 25, 1997

This manual copyright 1997 PERFORCE Software.
All rights reserved.

PERFORCE software and documentation is available from `http://www.perforce.com/`. You may download and use PERFORCE programs, but you may not sell or redistribute them. You may download and print the documentation, but you may not sell or redistribute it. You may not modify or attempt to reverse engineer the programs.

PERFORCE programs and documents are available from our Web site *as is*. No warranty or support is provided. Warranties and support, along with higher capacity servers, are sold by PERFORCE Software.

PERFORCE Software assumes no responsibility or liability for any errors or inaccuracies that may appear in this book.

By downloading and using our programs and documents you agree to these terms.

PERFORCE and *Inter-File Branching* are trademarks of PERFORCE Software. PERFORCE software includes software developed by the University of California, Berkeley and its contributors.

All other brands or product names are trademarks or registered trademarks of their respective companies or organizations.

The remainder of this column is left blank for your use. You may utilize it to store such written, inscribed, drawn, painted, copied, drafted, or duplicated material as you see fit, or for any other use, providing the use does not violate the laws, ordinances, statutes, regulations, edicts, canons, or decrees of your country, state, territory, kingdom, province, county, city, or other municipality. PERFORCE Software will assume no liability, responsibility, debt, risk or other obligation for any defamatory, libelous, pejorative, unlawful, slanderous, or otherwise illegal material appearing below this paragraph in this column on this page.

PREFACE

About This Manual

This is the *PERFORCE 97.3 User's Guide*. It teaches the use of PERFORCE's command-line interface; the PERFORCE Windows GUI is not discussed. For documentation on the Windows GUI interface, P4WIN, please see our *P4 to P4WIN Translation Guide* (for experienced PERFORCE users) or the *P4WIN User's Guide* (for the PERFORCE novice).

Although this guide can be used as a reference manual, it is primarily intended as guide/ tutorial on using PERFORCE. The full syntax of most of the PERFORCE commands is not provided here; we suggest that you supplement use of this guide with the upcoming *PERFORCE Command Reference*, or with the on-line help system. Use of this manual for operating systems other than UNIX and NT should be supplemented with the release notes for that OS.

New 97.3 Features

The features new to PERFORCE 97.3 have been marked with changebars in the left margin. The release notes provide more detailed information, and are available from our web site.

Please consult the release notes before upgrading from earlier versions of PERFORCE to version 97.3! The journaling and checkpointing subsystems have changed; the release notes contain safety instructions on performing the upgrade.

Margin Note Icons

This manual makes use of notes in the left margin to supply additional information. The icons accompanying these notes have the following meanings:

95/NT	Information specific to the Windows 95 or Windows NT command line.
✣	A cross-reference to other material in this manual.
🖥	A concrete example of the material discussed.

A note of general interest.

This note is rather important!

The Example Set

We have attempted to develop a uniform example set for use with this manual. All of the examples use the source code for `Elm`, a popular UNIX mail program. We selected the Elm source code for a number of reasons:

- Elm is widely used, and many PERFORCE users will be familiar with the program. If they are not, they at least understand what it does.
- The source code is stored in well-organized subdirectories, which allow us to demonstrate certain capabilities of PERFORCE.
- The source code for Elm is widely available; users of this manual can download Elm and try the examples as they're encountered.

Links to the Elm source code can be found at

```
http://www.myxa.com/elm.html
```

We are using the Elm source with the kind permission of Sydney Weinstein and Bill Pemberton of the USENET Community Trust.

Disclaimer: To the best of our knowledge, the Elm team has never used PERFORCE for source management. As far as we know, they never heard of PERFORCE until they received our email asking for permission to use their code in our manual. No implication that the Elm team uses or endorses PERFORCE is intended; none should be inferred.

Please Give Us Feedback

We are always interested in receiving feedback on our manuals. Dodoes this guide teach the topic well? Are there any glaring errors? Are the explanations clear, or are the exemplifications obfuscated by this enchiridion? Please let us know what you think; we can be reached at `manual@perforce.com`.

Table of Contents

CHAPTER 1

PERFORCE Concepts

You don't need to read this chapter if you don't want to.

All the material discussed here is also covered in the 'how-to' chapters, which comprise the rest of the manual.

This chapter is provided as a guide to what PERFORCE does, without the details of how to do it.

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

- PERFORCE offers version control: multiple revisions of the same file are stored, and older revisions are always accessible.
- PERFORCE provides facilities for concurrent development; multiple users can edit their own copies of the same file.
- Some release management facilities are offered; PERFORCE will track which revisions of which files are part of a particular release.
- Bugs and system improvement requests can be tracked from entry to fix; this is known as defect tracking.
- PERFORCE supplies some lifecycle management functionality; files can be kept in release branches, development branches, or in any sort of needed file set.
- Change review functionality is provided by PERFORCE; this allows users to be notified by email when particular files are changed.
- Although a build management tool is not built into PERFORCE, we do offer a companion freeware product called "`JAM - Make(1) Redux`". JAM and PERFORCE meet at the file system; source files managed by PERFORCE are easily built by `JAM`.

PERFORCE excels at all file management functions. Although PERFORCE was built to manage source files, it can manage any sort of on-line documents. It can be used to store revisions of a manual, to manage Web pages, or to store old versions of operating system administration files. Its branching functionality, which allows copies of files to evolve separately from the files they were copied from, is unparalleled in the industry. And PERFORCE is extremely fast.

PERFORCE Architecture

PERFORCE has a client/server architecture, in which many computers, called *clients*, are connected to one central machine, the *server*. Each user works on a client; at their command, files they've been editing are transferred to and from the server. The clients communicate with the server via TCP/IP.

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

Two programs do the bulk of PERFORCE's work:

The PERFORCE installation guide can be found in Chapter 13 .

- The P4D program is run on the PERFORCE server. It manages the shared file repository, and keeps track of users, clients, protections, and other PERFORCE metadata.

  P4d must be run on a UNIX or Windows/NT host.
- The P4 program is run on each PERFORCE client. It sends the users' requests to the P4d server program for processing, and communicates with P4D via TCP/IP.

  P4 client programs can be run on many platforms, including UNIX, Windows, VMS, Macintosh, BeOS, and Next hosts.

Moving Files Between the Clients and the Server

Basic PERFORCE usage is taught in Chapters 3 and 4.

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

The details of changelists are discussed in Chapter 7.

Files that have been edited within a client workspace are sent to the depot via a *changelist*, which is a list of files, and instructions that tell the depot what to do with those files. For example, one file might have been changed in the client workspace, another added, and another deleted. These file changes might be sent to the depot in a single changelist, which is processed *atomically*: either all the changes are made to the depot at once, or none of them are. This allows problem fixes that span multiple files to be updated in the depot at exactly the same time.

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

File Conflicts

Resolving file conflicts is the topic of Chapter 5.

When two users edit the same file, it is possible for their changes to 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; subsequently, the second user tries to do the same thing. If PERFORCE were to unquestioningly accept the second user's file into the depot, the first user's changes would not be included in the latest revision of the file (known as the *head revision*).

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

Labeling Groups of Files

Chapter 8 discusses labels.

It is often useful to mark a particular set of file revisions for later access. For examples, 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 single mnemonic name, like `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 the old file revisions into a client workspace.

Branching Files

The workings of Inter-File Branching is covered in Chapter 9.

Thus far, it has been assumed that all changes of files happen linearly. But this is not always the case: suppose that one source file needs to evolve in two separate directions; perhaps one set of upcoming changes will allow the program to run under VMS, and another set will make it a Mac program. Clearly, two separately evolving copies of the same files are necessary.

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

We're particularly proud of PERFORCE's branching mechanism. Most SCM systems allow some form of branching, but PERFORCE's is particularly flexible and elegant.

Job Tracking

You'll learn how to do job tracking in Chapter 10.

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

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

PERFORCE's job tracking mechanism does not implement all functionality normally supplied by full-scale defect tracking systems. Its simple functionality can be used as is, or it can be integrated with a full-scale job tracking system with a scripting language such as Perl.

Change Review and Daemons

Chapter 11 discusses change review, and contains instructions on creating your own daemons.

PERFORCE's *change review* mechanism allows users to receive email notifying them when particular files have been updated in the depot. The files that a particular user receives notification on is determined by that user. Change review is implemented by an external Perl program, or *daemon*, and can be recoded by a knowledgeable user, allowing change review functionality to be customized.

Protections

PERFORCE's protection mechanism is described in Chapter 14.

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

Permissions are granted or denied based on the user's username and IP address. Since PERFORCE usernames are easily changed, protections at the user level provide safety, not security. Protections at the IP address level are as secure as the host itself.

CHAPTER 2

Connecting to the P4D Server

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

This chapter assumes that both the P4D and P4 programs have been installed by a system administrator. Installation instructions can be found in Chapter 13.

Two programs do the bulk of PERFORCE's work:

- The P4D program is run on the PERFORCE server. It manages the shared file repository, and keeps track of users, clients, protections, and other PERFORCE metadata.
- The P4 program is run on each PERFORCE client. It sends the users's requests to the P4D server program for processing, and communicates with p4d via TCP/IP.

Each P4 program needs to know the address and port of the P4D server that it communicates with. This address is stored in the P4PORT environment variable.

Verifying the Connection to the P4D Server

A **p4** client needs to know two things in order to talk to the P4D server:

- The name of the host that P4D is running on
- The port that P4D is listening on.

These are set via a single environment variable, P4PORT. It is possible that your system administrator has already set P4PORT; if not, you'll need to set it yourself.

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

Example: Output from p4 info *when correctly connected to the P4D server*

```
User name: edk
Client name: eds_elm
Client unknown.
Current directory: /usr/edk
Client address: 206.14.52.194:3119
Server address: margarine:1818
Server root: /usr/local/p4root
Server version: P4D-FREEBSD (12/13/96)
Server license: test 10 users
```

The `server address:` field shows which P4D server has been connected to; it displays the host and port number that P4D is listening on.

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

Example: Error from `p4 info;` *the P4D server connection is incorrectly specified*

```
Error:
Connect to server failed; check P4PORT.
   TCP connect to perforce:1666 failed.
   perforce: host unknown.
```

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

Telling P4 Where P4D is

Before continuing, you'll need to ask your system administrator the name of the host that P4D is located on, and the number of the TCP/IP port it's listening on. Once you've obtained this information, set your `P4PORT` environment variable to *`host:port#`*, where *`host`* is the name of the host that P4D is running on, and *`port#`* is the port that `p4d` is listening on. For example:

If the P4D host is named...	**and the P4D port is named...**	**set P4PORT to:**
`dogs`	`3435`	`dogs:3435`
`x.com`	`1818`	`x.com:1818`

The definition of `P4PORT` can be shortened if P4 is running on the same host as P4D. In this case, only the P4D port number need be provided to P4. And if P4D is running on a host named or aliased `perforce`, listening on port `1666`, the definition of `P4PORT` for the P4 client can be dispensed with altogether. For example:

If the P4D host is named...	**and the P4D port is...**	**set P4PORT to...**
<same host as the `p4` client>	`9783`	`9783`
`perforce`	`1666`	<no value needed>

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

CHAPTER 3

PERFORCE Basics: Quick Start

This chapter teaches basic PERFORCE usage. You'll learn how to move files to and from the common file repository, how to back out of these operations, and the basic PERFORCE reporting commands.

These concepts and commands are painted with very broad strokes in this chapter; the details are provided in the next.

Underlying Concepts

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

PERFORCE was written to be as unobtrusive as possible; very few changes to your normal work habits are required. Files are still created in your own directories with a standard text editor; PERFORCE commands supplement your normal work actions instead of replacing them.

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

File Configurations Used in the Examples

The use of the Elm source code set is described in the `About This Manual` *chapter (page 3).*

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

A single depot is used to store the elm files, and perhaps other projects as well. The elm files will be shared by storing them under an `elm` subdirectory within the depot.

Each user will store his or her client workspace Elm files in a different subdirectory. The two users we'll be following most closely, Ed and Lisa, will work with their Elm files in the following locations:

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

Setting Up a Client Workspace

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

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

Naming the Client Workspace

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

Example: Naming the client workspace

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

```
$ export P4CLIENT=eds_elm
```

Each operating system or shell has its own method of defining environment variables; Appendix A describes how to create environment variables within each shell and OS.

Describing the Client Workspace to the PERFORCE Server

Many P4 commands, including `p4 client`*, display a form for editing in a standard text editor. The editor that is used is defined through the* `EDITOR` *environment variable.*

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

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

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

Example:
Setting the client root and the client view

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

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

If he were to leave the form as is, all of the files under `/usr/edk` *would be mapped to the depot, and they would map to the entire depot, instead of to just the* `elm` *project. He changes the values in the* `Root:` *and* `View:` *fields as follows:*

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

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

When Ed's done, he quits from the editor, and the `p4 client` *command completes.*

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

The read-only `Client:` field contains the string stored in the `P4CLIENT` environment variable. `Description:` can be filled with anything at all (up to 128 characters); this provides an arbitrary textual description of what's contained in this client workspace. The `View:` describes the relationship between files in the depot and files in the client workspace.

Creating a client specification has no immediate visible effect; no files are created when a client specification is created or edited. The client specification simply indicates where files will be located when subsequent P4 commands are used.

Editing an Existing Client Specification

`p4 client` can be used at any time to change the client workspace specification. Just as when a client specification is created, changing a specification has no immediate affect on the locations of any files; the location of files in the depot and workspace is affected only when the client specification is used in subsequent commands. But there is an important distinction between changing the client's root and changing the client's view: if you change the root, PERFORCE assumes that you will manually relocate the files as well. If you change the view and then bring files into the client from the depot, PERFORCE will delete and add files as necessary to make the client workspace reflect the view.

Deleting an Existing Client Specification

An existing client workspace specification can be deleted with `p4 client -d` *client-name*. Deleting a client specification has no effect on any files in the client workspace or depot; it simply removes the P4D server's record of the mapping between the depot and the client workspace. To delete existing files from a client workspace, use `p4 sync #none` (described on page 35) on the files *before* deleting the client specification, or use the standard local OS deletion commands *after* deleting the client specification.

Copying Files from the Workspace to the Depot

If you're working in an already-established PERFORCE environment, and want to start by retrieving already-existing files, you can skip to page 24 and come back to this section later.

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

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

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

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

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

Adding Files to the Depot

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

Example: Adding files to a changelist

Ed is writing a help manual for Elm. The files are named `elm-help.0` *through* `elm-help.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 elm-help.*
//depot/elm_proj/doc/elm-help.0#1 - opened for add
//depot/elm_proj/doc/elm-help.1#1 - opened for add
//depot/elm_proj/doc/elm-help.2#1 - opened for add
//depot/elm_proj/doc/elm-help.3#1 - opened for add
```

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

Example: Submitting a changelist to the depot

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

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

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

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

Multiple file arguments can be provided on the command line.

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

Ed wants to add all 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.>
```

... and then does a `p4 submit.`

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

Please see Chapter 5 for instructions on resolving file conflicts.

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

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

Updating Depot Files

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

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

Since the files must have their `write` permission turned back on before they can be edited, the `p4 edit` command must be given before the file is actually edited.

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

Example: Ed wants to make changes to his `elm-help.3` *file. He opens the file for edit:*

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

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

Deleting Files From the Depot

Files are deleted from the depot similarly to the way they are added and edited: the `p4 delete` command opens the file for delete in the default changelist, and then `p4 submit` is used to delete the file from the depot. `p4 delete` also deletes the file from the client workspace; this occurs when the `p4 delete` command is given. In essence, `p4 delete` replaces the operating systems `rm` or `del` command.

Example:
Deleting a file from the depot.

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

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

The file is deleted from the client workspace immediately; it is not deleted from the depot until the `p4 submit` *command is given.*

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

Submitting with Multiple Operations

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

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

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

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

He edits the existing files and then `p4 submit`*'s the default changelist:*

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

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

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

`p4 sync` *was named* `p4 get` *in previous versions of* P*ERFORCE*. `p4 get` *can still be used as an alias for* `p4 sync`.

Retrieving Files from the Depot into a Workspace

Files can be retrieved from the depot into a client workspace from the depot with `p4 sync`.

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

Example: Retrieving files from the depot into the client workspace.

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

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

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

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

Reverting Files to their Unopened States

Any file opened for `add`, `edit`, or `delete` can be removed from its changelist with `p4 revert`. This command will revert the file in the client workspace back to its unopened state.

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

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

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

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

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

If `p4 revert` is used on a file that had been opened with `p4 delete`, it will appear back in the client workspace immediately. If `p4 add` was used to open the file, `p4 revert` removes it from the changelist, but leaves the client workspace file intact. If the reverted file was originally opened with `p4 edit`, the last synced version will be written back to the client workspace, overwriting the newly-edited version of the file. In this case, you may want to save a copy of the file before running `p4 revert`.

Basic Reporting Commands

PERFORCE provides some 20+ reporting commands. Each chapter in this manual ends with a description of the reporting commands relevant to the chapter topic. All the reporting commands are discussed in greater detail in the reporting chapter, chapter 12.

The most basic PERFORCE 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 a few other tidbits.

Two other reporting commands are used quite often:

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

CHAPTER 4

PERFORCE Basics: The Details

The Quick Start chapter explained the basics of using PERFORCE, but discussion of the practical details were deferred. This chapter, which supplements the *Quick Start* chapter, covers the dry PERFORCE rules. The topics discussed include views, mapping depots to client workspaces, PERFORCE wildcards, rules for referring to older file revisions, file types, and form syntax.

It is assumed that the material in the *Quick Start* chapter has been read and properly digested.

Description of the Client Workspace

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

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

PERFORCE manages the files in a client workspace in a few direct ways. It creates, updates, or deletes files when the user requests PERFORCE to synchronize the client workspace with the depot; it turns on write permission when the user requests to edit a file; and turns off write permission and submits updated versions back to the depot when the user is finished editing the file.

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

PERFORCE's management of a client workspace requires a certain amount of cooperation from the user. Since client files are just plain files with write permission turned off, willful users can circumvent the system by turning on write permission, directly deleting or renaming files, or otherwise modifying the file tree supposedly under PERFORCE's control. PERFORCE counters this with two measures: first, PERFORCE has explicit commands to verify that the client workspace state is in accord with the server's recording of that state;

second, PERFORCE tries to make using PERFORCE at least as easy as circumventing it. For example: to make a temporary modification to a file, it is easier to use PERFORCE than it is to copy and restore the file manually.

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

In addition to accessing the client files, the P4 client program sometimes creates temporary files on the client host. Otherwise, PERFORCE neither creates nor uses any files on the client host.

Wildcards

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

Wildcard	**Meaning**
`*`	Matches anything except slashes, matches only within a single directory.
`...`	Matches anything including slashes; matches across multiple directories
`%d`	Used for parametric substitution in views. See page 32 for a full explanation

The '`...`' wildcard is passed by the P4 client program to the P4D server, where it is expanded to match the corresponding files known to P4D. The `*` wildcard is expanded locally by the OS shell before the P4 command is sent to the server, and the files that match the wildcard are passed as multiple arguments to the P4 command. To have PERFORCE match the `*` wildcard against the contents of the depot, it must be escaped, usually with quotes or a backslash. Most command shells don't interfere with the other two wildcards.

Wildcards and 'p4 add'

There is one case in which the "`...`" wildcard cannot be used, and that is with the `p4 add` command. The "`...`" wildcard is expanded by the P4D server, and since the server doesn't know what files are being added (after all, they're not in the depot yet), it can't expand that wildcard. The `*` wildcard *can* be used with `p4 add`; in this case, it is expanded by the local OS shell, not by the P4D server.

Mapping the Depot to the Client Workspace

Just as a client name is nothing more than an alias for a particular directory on the client machine, a depot name is an alias for a directory on the PERFORCE server. The relationship between files in the depot and files in the client workspace is described in the *client view*, and it is set with the `p4 client` command. When `p4 client` is typed, a variation of the following form is displayed:

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

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

Using Views

95/NT

On NT, the PERFORCE client workspace must be specified in a slightly different format if it is to span multiple drives. Please see the release notes, which are available at our Web site, for details.

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

```
//depotname/file_specification
```

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

```
//clientname/file_specification
```

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

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

- *The client view determines which files in the depot can be seen by a client workspace.* This is determined by the sum of the depot sides of the mappings within a view. A view might allow the client workspace to retrieve every file in the depot, or only those files within two directories, or only a single file.
- *It constructs a one-to-one mapping between files in the depot and files in the client workspace*. Each mapping within a view describes a subset of the complete mapping. The one-to-one mapping might be straightforward; for example, the client workspace file tree might be identical to a portion of the depot's file tree. Or it can be oblique; for example, a file might have one name in the depot and another in the client workspace, or be moved to an entirely different directory in the client workspace. No matter how the files are named, there is always a one-to-one mapping.

To determine the exact location of any client file on the host machine, substitute the value of the `p4 client` form's `Root:` field for the client name on the client side of the mapping. For example, if the `p4 client` form's `Root`: field for the client `eds_elm` is set to `/`

`usr/edk/elm`, then the file `//eds_elm/doc/elm-help.1` will be found on the client host in `/usr/edk/elm/doc/elm-help.1`.

On NT, the PERFORCE *client workspace must be specified in a slightly different format if it is to span multiple drives. Please see the release notes for details.*

Wildcards in Views

Any wildcard used on the depot side of a mapping must be matched with an identical wildcard in the mapping's client side. Any string matched by the wildcard will be 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` will be mapped to the client workspace file `//eds_elm/nls/gencat/README`.

Types of Mappings

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

Direct Client-to-Depot Views

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

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

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

Mapping the Full Client to only Part of the Depot

Usually only a portion of the depot is of interest to a particular client. The left-hand side of the `View` field can be changed to point to only the portion of the depot that's relevant.

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/...
```

Mapping Files in the Depot to a Different Part of the Client

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

Example:
Multiple mappings in a single client view.

The `elm_proj` *subdirectory of the depot contains a directory called* `doc,` *which has all the Elm documents. Included in this directory are four files named* `elm-help.0` *through* `elm-help.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`*; it's located at the same level as his* `doc` *directory. The four* `elm-help` *files will go here; he fills in the* `View` *field of the* `p4 client` *form as follows:*

```
//depot/...                           //mike_elm/...
//depot/elm_proj/doc/elm-help.*     //mike_elm/help/elm-help.*
```

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

Excluding Files and Directories from the View

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

Example:
Using views to exclude files from a client workspace

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

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

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

Allowing Filenames in the Client to be Different than Depot Filenames

Mappings can be used to make the names of files different in the client workspace than they are 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 `elm-help.`*X files in the depot and call them* `helpfile.`*X in his client workspace. He uses the following mappings:*

```
//depot/elm_proj...                 //mike_elm/...
//depot/elm_proj/doc/elm-help.*  //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 can be used to rearrange the order of the matched substrings.

Example: Changing string order in client workspace names

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

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

Two Mappings Can Conflict and Fail

It is possible for multiple mappings in a single view to lead to a situation in which the name does not map the same way in both directions. When a file doesn't map the same way in both directions, the file is ignored.

Example: Mappings that fail.

Joe has constructed a view as follows:

```
//depot/elm_proj/...    //joe/elm/...
//depot/nowhere/*    //joe/elm/doc/*
```

The depot file `//depot/elm_proj/doc/help` *would map to* `//joe/elm/doc/help`*, but the same file in the client workspace would map back to the depot via the higher-precedence second line to* `//depot/nowhere/help`*. Because the file would be written back to a different location in the depot than where it was read from,* P*ERFORCE doesn't map this name at all.*

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

Referring to Files on Command Lines

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

Local Syntax

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

PERFORCE Syntax

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

Examples of PERFORCE Syntax

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

PERFORCE syntax is sometimes called *depot syntax* or *client syntax*, depending on whether the file specifier refers to a file in the depot or on the client. But the syntax is the same in either case.

Multiple depots can be provided within a single PERFORCE server. See chapter 15 for details.

The specifier `//...` is occasionally used; it means ‘all files in all depots’.

Providing Files as Arguments to Commands

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

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

Example:
Uses of different syntaxes to refer to a file

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

- *While in his client root directory, he could type*

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

- *While in the* `src` *subdirectory, he could type*

  ```
  p4 delete lock.c
  ```

- *While in any directory on the client host, he can type*

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

  or

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

  or

  ```
  p4 delete /usr/edk/elm/src/lock.c
  ```

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

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

Wildcards and PERFORCE Syntax

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

`J*`	Files in the current directory starting with `J`
`*/help`	All files called `help` in current subdirectories
`...`	All files under the current directory and its subdirectories
`.../*.c`	All such files ending in `.c`
`/usr/edk/...`	All files under `/usr/edk`
`//weasel/...`	All files on client `weasel`
`//depot/...`	All files in the depot

Name and String Limitations

File Names

Because of PERFORCE's naming conventions, certain characters cannot be used in file names. These include unprintable characters, the above wildcards, and the PERFORCE revision characters `@` and `#`.

Descriptions

Label, branch, user, and client workspace specifications have a silent limit of 128 bytes on descriptions. The description field of a changelist or a job can be any length.

Specifying Older File Revisions

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

Change numbers are explained in chapter 7.

Labels are explained in chapter 8

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

`file@n`	A change number	`p4 sync lock.c@126`
		Refers to the version of `lock.c` when changelist 126 was submitted, even if it was not part of the change.
		
		`p4 sync //depot/...@126`
		Refers to the state of the entire depot at changelist 126

`file@label`	A label name	`p4 sync lock.c@beta` The revision of `lock.c` in the label called `beta`
`file@client-name`	A client name. The revision of *file* last taken into client workspace *clientname*.	`p4 sync lock.c@lisag_ws` The revision of `lock.c` last taken into client workspace `lisag_ws`
`file``#none`	The nonexistent revision.	`p4 sync lock.c#none` Says that there should be no version of `lock.c` in the client workspace, even if one exists in the depot.
`file``#head`	The head revision, or latest version, of the file.	`p4 sync lock.c#head` Except for explicitly noted exceptions, this is identical to referring to the file with no revision specifier.
`file``#have`	The revision on the current client. This is synonymous to `@client` where client is the current client name.	`p4 sync lock.c#have` The revision of `lock.c` found in the current client.

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

Using Revision Specifications without Filenames

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

Example: Retrieving files using revision specifiers

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

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

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

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

He could have typed

```
p4 sync @eds_elm
```

and the effect would have been the same.

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

Another client needs all its files removed, but wants PERFORCE *to know that it still contains those files. Ed sets* `P4CLIENT` *to the correct clientname, and types*

```
p4 sync #none
```

Revision Ranges

A few PERFORCE client commands can limit their actions to a range of revision numbers, rather than just a single revision. A revision range is two revision specifications, separated by a comma. If only a single revision is given where a revision range is expected, the named revision specifies the end of the range, and the start of the range is assumed to be 1. If no revision number or range is given where a revision range is expected, the default is all revisions.

File Types

PERFORCE supports normal text files as well as binary, "large text" files, keyword text files, Macintosh resource forks, and symbolic links. PERFORCE attempts to determine the type of the file automatically: when a file is opened with `p4 add`, PERFORCE first determines if the file is a regular file or a symbolic link, and then examines the first part of the file to determine whether is it 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.

The detected file type can be overridden with `p4 add -t` *`type`*, where *`type`* is one of `binary`, `text`, `ltext`, `xbinary`, `xtext`, `ktext`, `kxtext`, `resource` or `symlink`. Descriptions of these file types are provided below.

A file's type is inherited from one revision to the next. A file can be opened with a different type for the new revision with `p4 edit -t` *`type`*. If a file has already been opened with `p4 add` or `p4 edit`, its type can be changed with `p4 reopen -t` *`type`*.

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

PERFORCE must sometimes store the complete version of every file in the depot, but most often it stores only the changes in the file since the previous revision. This is called *delta storage*, and PERFORCE uses RCS format to store its deltas. The file's type determines whether *full file* or *delta* storage is used. When delta storage is used, file merges and file compares can be performed. Files that are stored in their full form can't be merged or compared.

The PERFORCE file types are:

Keyword	**Description**	**Comments**	**Storage Type**
`text`	Text file	Treated as text on the client	`delta`
`xtext`	Executable text file	Like a `text` file, but execute permission is set on the client file.	`delta`
`binary`	Non-text file	Accessed as binary files on the client	`full file`
`xbinary`	Executable binary file	Like a `binary` file, but execute permission is set on the client file.	`full file`
`ltext`	Long text file	This type should be used for generated text files, such as PostScript files.	`full file`
`symlink`	Symbolic link	UNIX clients access these as symbolic links; non-UNIX clients treat them as (small) `text` files	`delta`
`ktext`	Text file with keyword expansion.	Any inclusion of the literal string `$Id$` within the file will be expanded to reflect the depot file name and revision number.	`delta`
`kxtext`	Executable text file with keyword expansion	Like a `ktext` file, but execute permission is set on the client file	`delta`
`resource`	Macintosh resource fork	Please see the Macintosh client release notes at `<http://www.perforce.com/perforce/doc/macnotes.txt>`	`full file`

To find the type of an existing file, use the `p4 opened` reporting command.

Forms and PERFORCE Commands

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

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

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

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 these commands can take file name arguments, with or without wildcards, to limit reporting to specific files. Without the file arguments, the reports are generated for all files.

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

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

Revision specifiers can be used with all of these reporting commands, for example

```
p4 files @clientname
```

can be used to report on all the files in the depot that are currently found in client *clientname.*

Chapter 12, the reporting chapter, contains a more detailed discussion of each of these commands.

CHAPTER 5

PERFORCE Basics: Resolving File Conflicts

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

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

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

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

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

RCS Format: How PERFORCE Stores File Revisions

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

Only the Differences Between Revisions are Stored

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

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

foo#3:

This is a test
of the
emergency
broadcast system

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

foo#2:

line 3 was “urgent”

And revision 1 would be a representation of this:

foo#1:

line 4 was “system”

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

This is a test
of the
urgent
system

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

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

Use of 'diff' to Determine File Revision Differences

RCS utilizes the 'GNU `diff`' program to determine the differences between two versions of the same file; P4D contains its own *diff* routine which is used by PERFORCE servers to determine file differences when storing deltas. Because PERFORCE's *diff* always determines file deltas by comparing chunks of text between newline characters, it is by default only used with text files. If a file is binary, each revision is usually stored in full, but binary files can be checked in as text files, insuring that only the deltas are stored.

Scheduling Resolves of Conflicting Files

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

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

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

Why 'p4 sync' to Schedule a Resolve?

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

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

Example: Scheduling resolves with `p4 sync`

Ed is making a series of changes to the `*.guide` *files in the elm* `doc` *subdirectory. He has retrieved the* `//depot/elm_proj/doc/*.guide` *files into his client and has opened the files with* `p4 edit`*. He edits the files, but before he has a chance to submit them, Lisa submits new versions of some of the same files to the depot. The versions Ed has been editing are no longer the head revisions; resolves must be scheduled and performed for each of the conflicting files before Ed's edits can be accepted. Ed schedules the resolves with* `p4 sync //edk/doc/*.guide`*. Since these files are already open in the client, PERFORCE doesn't replace the client files; instead, PERFORCE schedules resolves between the client files and the head revisions in the depot.*

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

How Do I Know When a Resolve is Needed?

`p4 submit` will fail if it determines that any of the files in the submitted changelist need to be resolved, and the error message will include the names of the files that need resolution. If the changelist provided to `p4 submit` was the default changelist, it will be assigned a number, and this number must be used in all future references to the changelist.

Please see Chapter 7 for a discussion of numbered changelists.

Another way of determining whether a resolve is needed is to run `p4 sync -n` *`filenames`* before performing the submit, using the files in the changelist as arguments to the command. If file conflict resolutions are necessary, `p4 sync -n` will report them. The only advantage of this scheme over viewing the submit error is that the default changelist will not be assigned a number.

Performing Resolves of Conflicting Files

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

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

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

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

File Revisions Used and Generated by 'p4 resolve'

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

`yours`	The newly-edited revision of the file in the client workspace. This file is overwritten by *result* once the resolve process is complete.
`theirs`	The revision in the depot that the client revision conflicts with. Usually, this is the head revision, but `p4 sync` can be used to schedule a resolve with any revision between the head revision and *base*.
`base`	The file revision in the depot that *yours* was edited from. Note that *base* and *theirs* are different revisions; if they were the same, there would be no reason to perform a resolve.

`merge`	File variation generated by PERFORCE from *theirs*, *yours*, and *base*.
`result`	The file resulting from the resolve process. *result* is written to the client workspace, overwriting *yours*, and must subsequently be submitted by the user. The instructions given by the user during the resolve process determine exactly what is contained in this file. The user can simply accept *theirs*, *yours*, or *merge* as the result, or can edit *merge* to have more control over the result.

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

Discussion of resolving branched files begins on page 47.

Types of Conflicts Between File Revisions

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

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

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

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

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

How the Merge File is Generated

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

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

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

The `merge` *file is generated by P4D's internal* diff *routine. But the differences displayed by* `dy`, `dt`, `dm`, *and* `d` *are generated by a* diff *internal to the P4 client program, and this* diff *can be overridden by specifying an external* diff *in the* `P4DIFF` *environment variable.*

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*
`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 `MERGE` *base theirs yours merge* To use this option, you must set the environment variable MERGE to the name of a third-party program that merges the first three files and writes the fourth as a result.
`?`	help	Display help for `p4 resolve`
`s`	skip	Don't perform the resolve right now.
`ay`	accept yours	Accept *`yours`* into the client workspace as the resolved revision, ignoring changes that may have been made in *`theirs`*.
`at`	accept theirs	Accept *`theirs`* into the client workspace as the resolved revision. The revision that was in the client workspace is trashed.
`am`	accept merge	Accept *merged* into the client workspace as the resolved revision. The version originally in the client workspace is trashed.
`a`	accept	Keep PERFORCE's recommended result. Depending on the circumstances, this will be either *`yours`*, *`theirs`*, or *`merge`*.

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 command line has the following format:

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

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

Example: Resolving file Conflicts

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

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

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

```
Intuitive Systems
Mountain View, California
>>>> ORIGINAL VERSION
==== THEIR VERSION
98992
==== YOUR VERSION
98993
<<<<
```

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

```
Intuitive Systems
Mountain View, California
98992
```

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

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

File locking is described in "Locking Files to Minimize File Conflicts", later in this chapter.

Using Flags with Resolve to Non-Interactively Accept Particular Revisions

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

`-ay`	Automatically accept *`yours`*.
`-at`	Automatically accept *`theirs`*. Use this option with caution; the file revision in the client workspace will be overwritten with no chance of recovery.
`-am`	Automatically accept the PERFORCE-recommended file revision (`yours`, `theirs`, or `merge`) if there are no conflicting line sets. If there are conflicts, skip the resolve for 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*; otherwise skip this file.

Example: Automatically accepting particular revisions of conflicting files

Ed has been editing the `doc/*.guide` *files, and knows that some of them will require resolving. He types* `p4 sync doc/*.guide`*; all of these files that conflict with files in the depot are scheduled for resolve. He then types* `p4 resolve -am`*; the merge files for all scheduled resolves are generated, and those merge files that contain no line set conflicts are written to his client workspace. He'll still need to manually resolve all the other conflicting files, but the amount of work he needs to do is substantially reduced.*

Binary Files and 'p4 resolve'

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

Locking Files to Minimize File Conflicts

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

The clear benefit of `p4 lock` is that once a file is locked, the user who locked it will experience no further conflicts on that file, and will not need to resolve the file. But this comes at a price: other users will not be able to submit the file until the file is unlocked, and will have to do their own resolves once they submit their revision. Under *most* circumstances, a user who locks a file is essentially saying to other users “I don’t want to deal with any resolves; *you* do them.” But there is an exception to this rule.

Preventing Multiple Resolves with File Locking

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

Ed opens file `foo` for edit;
Lisa opens the same file in her client for edit;
Ed and Lisa both edit their client workspace versions of `foo`;
Ed submits a changelist containing that file, and his submit succeeds;
Lisa submits a changelist with her version of the file; her submit
fails because of file conflicts with the new depot’s `foo`;
Lisa starts a resolve;
Ed edits and submits a new version of the same file;
Lisa finishes the resolve and attempts to submit; the submit fails and must now
be merged with Ed’s latest file.
<etc...>

File locking can be used in conjunction with resolves to avoid this sort of headache. The sequence would be implemented as follows: before scheduling a resolve, lock the file. Then sync the file, resolve the file, and submit the file. New versions can’t be submitted by other users until the resolved file is either submitted or unlocked.

Resolves and Branching

Files in separate codelines can be integrated with `p4 resolve`; discussion of resolving branched files begins in the *Branching* chapter on page 68.

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` will fail. 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 P4, 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.

Chapter 12 has a longer description of each of these commands; `p4 help` provides a complete listing of the many flags for these reporting commands.

CHAPTER 6

PERFORCE Basics: Miscellaneous Topics

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

- Command-line flags common to all PERFORCE commands;
- How to work on files while not connected to a PERFORCE server;
- Refreshing the client workspace;
- Additional `p4 client` options;
- Renaming files;
- Flags that allow form data to be read from a file;
- Recommendations for organization of files within the depot.

Command-Line Flags Common to All PERFORCE Commands

Five flags are available for use with all PERFORCE commands. These flags are given between the system command `p4` and the command argument taken by `p4`. These flags are:

Flag	Meaning	Example
`-c clientname`	Runs the command on the specified client. Overrides the `P4CLIENT` environment variable.	`p4 -c joe edit //depot/foo` Opens file `foo` for editing under client workspace `joe`.
`-d directory`	Specifies the current directory, overriding the environment variable `PWD`.	`p4 -d ~elm/src edit foo bar` Opens files `foo` and `bar` for edit; these files are found relative to `~elm/src`.
`-p server_addr`	Gives the `p4d` server's listening address, overriding `P4PORT`.	`p4 -p mama:1818 clients` Reports a list of clients on the server on host `mama`, port `1818`.

Flag	**Meaning**	**Example**
`-u username`	Specifies a PERFORCE user, overriding the `P4USER`, `USER`, and `USERNAME` environment variables.	`p4 -u bill user` Presents the `p4 user` form to edit specification for user `bill`. Only those commands that the specified user has permissions on may be run.
`-x filename`	Instructs `p4` to read arguments, one per line, from the named file.	*See* `Working Detached` *section, below*

All PERFORCE commands can take these flags, even commands for which this flag usage is clearly useless; e.g. `p4 -u bill -d /usr/joe help`. Other flags are available as well; these additional flags are command dependent. Please use `p4 help` *`commandname`* to find the flags available to each command.

Working Detached

Under normal circumstances, users work in their client workspace with a functioning network connection to a PERFORCE server. As they edit files, they are supposed to announce their intentions to the server with `p4 edit`, and the server responds by noting the edit in the depot's metadata, and by unlocking the file in the client workspace. However, it is not always possible for a network connection to be present; a method is needed for users to work entirely detached from the server

The scheme is as follows:

- The user works on files without giving PERFORCE commands; instead, OS commands are given that manually change the permissions on files, and then these files are edited or deleted.
- If the files were not edited within the client workspace, they should be copied to the client workspace when the network connection is reestablished.
- The `p4 diff` reporting program is used to find all files in the workspace that have changed without PERFORCE's knowledge. Output from this command is used to bring the depot in sync with the client workspace

Finding Changed Files with 'p4 diff'

It is perfectly safe to use `p4 edit` on any file; this command gives the local file `write` permissions, but does not otherwise alter it.

The `p4 diff` reporting command is used to compare a file in the client workspace with the corresponding file in the depot. Its behavior 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`.

Using ‘p4 diff’ to Update the Depot

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

To open changed files for edit after working detached, use

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

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

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

As always, these `edit` and `delete` requests are stored in a changelist, which is not processed until the `p4 submit` command is given.

`p4 sync -f` is similar to `p4 refresh`, a command found in previous versions of PERFORCE. Whereas `p4 refresh` only copied unopened files from the depot that PERFORCE thinks you already have, `p4 sync -f` copies any *unopened files from the depot to the client workspace.*

`p4 refresh` will still run in this version of PERFORCE, but will disappear in some future version.

Refreshing files

The process of syncing a depot with a formerly-detached client workspace has a converse: it is possible for PERFORCE to become confused about the contents of a client workspace through the *accidental* use of UNIX commands. For example, suppose that you accidently delete a client workspace file via the UNIX `rm` command, and that the file is one that you wanted to keep. Even after a submit, `p4 have` will still list the file as being present in the workspace.

Just as the process described above will bring the depot in sync with the client workspace, `p4 sync -f` *`files`* can be used to bring the client workspace in sync with the files the depot *thinks* you have. This command is mostly a recovery tool for bringing the client workspace back into sync with the depot after accidentally removing or damaging files managed by PERFORCE.

Options in the ‘p4 client’ Form

The form brought up by `p4 client` has an `option` field, which takes two values:

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

The ‘`modtime`’ option controls the modification times of client files when gotten from the depot with `p4 sync` or `p4 revert`. Setting this value to `modtime` leaves the modification times of files in the client as the times these files were submitted to the depot. If this option is left as the default, `nomodtime`, the modification date is set to the time the file was copied into the client.

The ‘`clobber`’ option, which can be set to `clobber` or `noclobber`, controls how `p4 sync` behaves while retrieving files from the depot that already exist in the client. The default, `noclobber`, tells `p4 sync` to avoid clobbering client files that aren’t open in PERFORCE but have otherwise been made writable by the user. If `clobber` is selected, these files will be overwritten.

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 would look like this:

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

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

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

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

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

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

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

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

Another way of solving the same problem would be to have the PERFORCE system administrator create one depot for each project or branch. Please see Chapter 15, page 109, for details.

Renaming Files

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

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

The *from_file* will be moved to the directory and renamed according to the *to_file* specifier. For example, if *from_file* is `d1/foo` and *to_file* is `d2/bar`, then `foo` will be moved to the `d1` directory, and will be renamed `bar`. The *from_file* and *to_file* specifiers may include wildcards, as long as they are matched on both sides. PERFORCE `write` access is needed on all the specified files.

PERFORCE access levels are explained in chapter 14.

Reading Forms from Standard Input; Writing Forms to Standard Output

Any commands that require the user to fill in a form, such as the `p4 client` and `p4 submit` commands, can read the form from standard input with the `-i` flag. An example is

```
p4 client -i < filename
```

where *filename* contains the field names and values expected by the form. Similarly, the `-o` flag can be used to write a form specification to standard output.

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

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

CHAPTER 7

Changelists

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

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

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

- Sometimes a user wants to split files into separate groups for submission. For example, suppose a user is fixing two bugs, each of which spans a separate set of files. Rather than submit the fixes to both bugs in a single changelist, the user might elect to create one changelist for the files that fix the first bug, and another changelist for the files that fix the second bug. Each changelist would be submitted to the depot via separate `p4 submit`'s.
- Under certain circumstances, the `p4 submit` command can fail. For example, if one user has a file locked and another user submits a changelist that contains that file, the submit will fail. When a submit of the default changelist fails, the changelist is assigned a number, is no longer the default changelist, and must be referred to by its number.

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

Working with the Default Changelist

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

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

Each of the files in the changelist are said to be *open* within the client workspace: the first of the files above was opened for edit with `p4 edit`, and the second was opened for deletion with `p4 delete`. The files in the changelist are updated within the depot with `p4 submit`, which sends the changelist to the server; the server processes the files contained in the changelist and alters the depot accordingly.

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

The commands that add or remove files from changelists are:

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

By default, these commands, and `p4 submit`, act on the default changelist; for example, if a user types `p4 add filename`, this file is added to the default changelist. When a user types `p4 submit`, the default changelist is submitted.

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

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

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

A PERFORCE superuser may change the description of a changelist, and in some cases may delete changelists entirely. Please see page 108 for details.

Creating Numbered Changelists Manually

A user can create a changelist in advance of submission with `p4 change`. This command brings up the same form seen during `p4 submit`. All files in the default changelist are moved to this new changelist; when the user quits from the form, the changelist is assigned the next changelist number in sequence, and this changelist must be subsequently referred to by this change number. Files can be deleted from the changelist by editing the form; files deleted from this changelist are moved to the next default changelist. The status for a changelist created by this method is `pending` until the form is submitted.

Any client file may be included in only one pending changelist.

Working With Numbered Changelists

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

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

Example: Working with multiple changelists

Ed is working on two bug fixes simultaneously. One of the bugs involves mail filtering and requires updates of files in the filter subdirectory; the other problem is in the elm aliasing system, and requires an update of `utils/elmalias.c`*. Ed wants to update each bug separately in the depot; this will allow him to refer to one bug fix by one change number and the other bug fix by another change number. He's already started fixing both bugs, and has opened some of the affected files for edit. He types* `p4 change`*, and sees*

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

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

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

When he quits from the editor, he's told

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

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

He fixes both bugs at his leisure. He realizes that the filter problem will require 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 cre-*

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

Automatic Creation and Renumbering of Changelists

When Submit of the Default Changelist Fails, the Changelist is Assigned a Number

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

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

  User A types `p4 edit //depot/foo`;
  User B types `p4 edit //depot/foo`;
  User B submits her default changelist;
  User A submits his default changelist.

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

If any file in a changelist is rejected for any reason, the entire changelist is backed out, and none of the files in the changelist are updated in the depot. If the submitted changelist was the default changelist, PERFORCE assigns the changelist the next change number in sequence, and this change number must be used from this point on to refer to the changelist.

Chapter 5 discusses the merge/resolve process.

If the submit failed because the client-owned revision of the file is not the head revision, a *merge* must be performed before the changelist will be accepted.

PERFORCE May Renumber a Changelist upon Submission

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

Example: Automatic renumbering of changelists

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

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

Deleting Changelists

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

Changelists that have already been submitted can never be deleted.

Changelist Reporting

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

Command	**Meaning**
`p4 changes`	Displays a list of all pending and submitted changelists, one line per changelist, and an abbreviated description.
`p4 changes -m` *`numchanges`*	Limits the number of changelists reported on to the last *numchanges* changelists.
`p4 changes -s` *`status`*	Limit the list to those changelists with a particular status; for example, `p4 changes -s submitted` will list only already-submitted changelists.
`p4 describe` *`changenum`*	Displays full information about a single changelist; if the changelist has already been submitted, the report will include a list of affected files and the diffs of these files. The `-s` flag can be used to exclude the diffs of the files.

CHAPTER 8

Labels

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

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

Create a label when:

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

Why Not Just Use Change Numbers?

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

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

Creating a Label

Labels are created with `p4 label` *`labelname`*.; this command brings up a form similar to the `p4 client` form. Like clients, labels have associated views; the label view limits which files can be referenced by the label. Once the label has been created, the `p4 labelsync` command is used to load the label with file references.

Label names share the same namespace as clients, branches, and depots; thus, a label name can't be the same as any existing client, branch, or depot name.

Example: Creating a label

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

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

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

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

The `View:` field is used to limit the files that are included in the label. These files must be specified by their location in the depot; this view differs from other views in that only the depot side of the view is specified. The `locked` / `unlocked` option in the `Options`: field can prevent `p4 labelsync` from overwriting previously synced labels (this is described further in "*Preventing Accidental Overwrites of a Label's Contents*" on page 62).

Adding and Changing Files Listed in a Label

Once a label has been created, references to files can be included in the label with the `labelsync` command. The syntax for `labelsync` is

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

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

1. All files listed in a label must be contained in the label view specified in the `p4 label` form. Any files or directories that are not mapped through the label view are ignored by `labelsync`. All the following rules assume this, without further mention.
2. When `labelsync` is used to include a particular file in a label's file list, the file is added to the label if it is not already included in the label. If a different revision of the file is already included in the label's file list, it is replaced with the newly-specified revision. Only one revision of any file is ever included in a label's file list.
3. If `labelsync` is called with no filename arguments, as in

   ```
   p4 labelsync -l labelname
   ```

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

Example: Storing file references in a label with `p4 labelsync`

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

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

and sees the following:

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

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

4. If `p4 labelsync` is called with filename arguments, and the arguments contain no revision specifications, the head revisions of these files are included in the label's file list.

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

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

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

and sees

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

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

5. If `labelsync` is called with filename arguments that contain revision specifications, these file revisions are included in the label's file list.

Example: Including a different file revision in a label

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

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

and sees

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

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

Previewing Labelsync's Results

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

Preventing Accidental Overwrites of a Label's Contents

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

Retrieving a Label's Contents into a Client Workspace

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

Example: Retrieving files into a client workspace from a label

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

```
p4 sync //depot/...@labelname
```

or even

```
p4 sync @labelname
```

But she's interested in seeing only the header files from that label; she types

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

and sees

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

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

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

Deleting Labels

A label can be deleted in its entirety with

```
p4 label -d labelname
```

Files can be deleted from labels with

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

A variant of this is

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

This command deletes all the files from the label's file list, but leaves the empty label in the system.

Label Reporting

The commands that output reports on labels are:

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

CHAPTER 9

Branching

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

What is Branching?

Branching is a method of keeping in sync two or more sets of similar, but not identical, files. Most software configuration management systems have some form of branching; we believe that PERFORCE's mechanism is unique in that 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 become 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 whenever two sets of code have different rules governing when code should be submitted, or whenever a set of files needs to evolve along different paths. For example:

- The members of the development group want to submit code to the depot whenever their code changes, whether or not it compiles; but the release engineers don't want code to be submitted until it's been debugged, verified, and signed off on. They would branch the release codeline from the development codeline; when the development codeline is ready, it would be integrated into the release codeline. Patches and bug fixes would be made in the release code; later, these changes could be integrated into the development code.

- A company is writing a driver for a new multi-platform printer. They've written a UNIX device driver; they're now going to begin work on a Macintosh driver, using the UNIX code as their starting point. They create a branch from the existing UNIX code; they now have two copies of the same code, and these codelines can evolve separately. If bugs are found in either codeline, bug fixes can be propagated from one codeline to the other with the PERFORCE `integrate` command.
- At PERFORCE, we use branching to manage our releases. Development always proceeds in files located within `//depot/main/...` When a new release is ready, it's branched into another codeline, for example, the code for this release was copied from `//depot/main/...` into `//depot/97.3/...` Bug fixes that affect both codelines will be made within `//depot/main/...`, and later integrated into the other codeline.

  Development of release `98.1` will proceed in `//depot/main/...`, when the new release is ready, it will be branched into `//depot/98.1/...`, and the process will continue like this for all PERFORCE releases.

Branching's First Action: Creating a Branch

As described above, two separate actions comprise branching: first, a branch is created (e.g., files are copied); second, edits are copied from one codeline to the other as needed. This section describes the first of these actions.

In previous versions of PERFORCE, it was necessary to perform the fifth step: retrieval of the new codeline files into the client workspace with `p4 sync`. *This step is no longer necessary;* `p4 integrate` *now copies the files into the client workspace for you. To disable this automatic file copying, use* `p4 integrate -v`*; if you do this, you'll need to perform a sync after the submit.*

The steps to creating a branched codeline are:

1. Create the new branch view with `p4 branch` *`branchname`*. Use the view in the resulting form to indicate which files are to be included in the branch, and where the branched codeline will be stored within the depot's file tree.
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` to open the new files for branching. The new files are listed in a changelist; the associated operation is `branch`.
4. Use `p4 submit` to submit the changelist to the PERFORCE server. This creates the new files in the depot.
5. ~~Copy the new files from the depot to the client workspace with `p4 sync`.~~

The following example demonstrates each of these steps.

Step 1: Create the Branch View

The first step is to create the branch view. Creating a branch view does four things:

1. Assigns the branched codeline a name;
2. Describes which files will be copied from;
3. For each original file, describes where the new copy will be stored within the depot;

4. Maintains a mapping between each original and branch file, so that changes to one can be easily propagated to the other.

Example: Creating a branch

A version of Elm is ready for release, and a potential problem is foreseen: the developers will be submitting code to the depot for the next version of Elm, but the release engineers will be submitting fixes to the released version. The two policies are clearly incompatible; so a branched codeline, with duplicate Elm files, needs to be created. Kurt, one of the release engineers, is assigned to create the branch for the release engineers.

The original code is stored in the depot under its `elm_proj` *subtree; Kurt decides to call the branch* `elm_r1`*, and will store the branched codeline in the depot under an* `elm_release1` *subdirectory. He types*

```
p4 branch elm_r1
```

and sees the following:

```
Branch: elm_r1
Date:   05/25/1997 17:43:28
Owner:  kurtv
Description:
        Created by kurtv.
View:
        //depot/... //depot/...
```

The default `View` *above would map the entire depot to itself in a branch, which is useless. The* `View` *needs to map the original codeline's files on the left to branch files on the right; Kurt changes the* `View` *field as follows:*

```
Branch: elm_r1
Date:   05/25/1997 17:43:28
Owner:  kurtv
Description:
        Created by kurtv.
View:
        //depot/elm_proj/... //depot/elm_release1/...
```

This maps all the files in the depot's `elm_proj` *file tree to a new depot file tree called* `elm_release1`*. All files from the source subtree will eventually be copied to the branch subtree; these files will be the contents of the branch.*

Kurt quits the editor; the branch is created.

The `p4 branch` command does not copy files into the branch; it simply specifies which original file will correspond to which branched file.

Exclusionary mappings may be used within a branch view.

Step 2: Include the Branched Files in the Client View

In order to work with branched files, the branched files must be accessible through the client view.

Example: Including branched files in a client view

Kurt will be working with the branched files. His client is `kurtv_cli`*; he types* `p4 client`*, and adds a line to his client view:*

```
Client: kurtv_cli
Date:   05/25/1997 18:34:58
Owner:  kurtv
Description:
        Created by kurtv.
Root:   /usr/kurtv
View:
        //depot/elm_release1/...   //kurtv_cli/elm.r1/...
```

There might be other mappings within the client view; the only crucial factor is that the files in the depot's elm branch directory be mapped to some location in Kurt's client workspace. The mapping shown here accomplishes this.

Steps 3 & 4: Use 'p4 integrate' and 'p4 submit' to Create the Target Files

To create the new branch files, use `p4 integrate` followed by `p4 submit`. When the branch files don't yet exist in the depot, `integrate` creates the branched files in the client workspace and tells the server that the branch files are to be copied from the original files described in the branch mapping. The `integrate` command, like `add`, `edit`, and `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 basic syntax of the `integrate` command is

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

If the filepatterns are left off, all the files in the branch are affected. When included, the filepatterns must describe the new files, not the original files, and these files must be visible through the client view.

Example: Using `integrate` *to create branched files*

Kurt has created the branch `elm_r1` *as above, and he's ready to create the branched copies in the depot. He types* `p4 integrate -b elm_r1`*, and sees*

```
//depot/elm_release1/Changes#1 - branch/sync from //depot/elm_proj/Changes#6
//depot/elm_release1/NOTICE#1 - branch/sync from //depot/elm_proj/NOTICE#23
   <etc.>
```

The branched files have been created in his client workspace, and instructions to branch these files have been added to his default changelist. He types `p4 submit`*, and sees*

```
Change: new
Client: kurtv_elm
User:   kurtv
Status: new
Description:
        <enter description here>
Files:
        //depot/elm_release1/Changes    # branch
        //depot/elm_release1/Configure  # branch
   <etc.>
```

He changes the description and quits the editor; the branched files are created within the depot and are copied into the client workspace.

If Kurt wanted the files to be created in the depot but not synced to the client workspace, he could have used the `-v` flag with the `integrate` command. If he did this, the files would later need to be copied to the client workspace with `p4 sync`.

Editing Newly Branched Files

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, simply `p4 edit` the file.

Working With Branched Files

Once a branch has been created and the files have been copied into the branched codeline with `p4 integrate`, the branched files are treated exactly like non-branched files, with the normal use of `sync`, `edit`, `delete`, `submit`, etc. Evolution of both codelines proceeds separately; additional PERFORCE commands are used only when changes to one codeline need to be propagated to the other.

Branching's Second Action: Propagating Changes from One Codeline to the Other

It is worth repeating that two separate actions comprise branching: first, one set of files is copied from one location in the depot to another location, and second, changes made to one codeline can be copied to the branched codeline *as needed*. The steps needed to accomplish the first action have been described above; now we'll discuss how to accomplish the second action.

Discussion of file conflict resolution begins on page 39.

Edits to a file in either codeline can be propagated to the corresponding file in the other codeline with the `resolve` command. Only one additional step needs to be performed: before resolving, the `integrate` command is used to schedule the merge between the original files and the branched files. In its normal use with branched files, `p4 integrate` takes the form `p4 integrate -b branchname` ***`files`***, where the specified *files* are the branched files rather than the original files.

Example: Propagating original codeline changes to the branched codeline

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

```
p4 integrate -b elm_r1 ~kurtv/elm.r1/src/elm.c
```

and sees

```
//depot/elm_release1/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/kurtv/elm.r1/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.*

In PERFORCE terminology, changes are always propagated from *donor* files to *target* files. In the above example, the original codeline provided the donor files and the target files were located in the branched codeline, but changes can be propagated in the other direction as well.

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. You must tell PERFORCE that text in one file needs to be propagated to its branch with `p4 integrate`. 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.

`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 donor files supplied as arguments to `integrate` need not be in the client view.

Protections are discussed in Chapter 9.

To run the `p4 integrate` command, `write` access is needed on the target files, and `read` access is required on the donor files.

Propagating Changes from Branched Files to the Original Files

A change can be propagated in the reverse direction, from branched files to the original files, by supplying the `-r` flag to `p4 integrate`. In this case, the names of the original files are provided as arguments to `p4 integrate -r`.

Example: Propagating branched codeline changes to the original codeline

Ed wants to integrate a change in Kurt's branched `src/screen.c` *file to Ed's original version of the same file. He types* `p4 integrate -r -b elm_r1 //depot/elm_proj/src/screen.c`*; and then* `p4 resolve`*. The change in the branched file is propagated to his source file.*

When the `-r` flag is used to propagate changes from branched donors to original targets, the original source files must be visible through the target view.

Branching and Merging Without a Branch View

Thus far, we have been describing the two actions that comprise branching: copying a set of files from one location in the depot to another, and propagating edits of one of the codelines to the other codeline. It is possible to use `p4 integrate` to perform both of these steps without ever having created a branch view. This is accomplished by calling `p4 integrate` with two file arguments and without the `-b branch` flag, as in

```
p4 integrate donor_file target_file
```

When `p4 integrate` is called this way, it will perform the integration between the two named files. The first file argument provides the donor; the second provides the target. The donor file must already exist; the target file needn't. There are three possible combinations of donor and target:

- *The target file doesn't exist*. In this case, the target file is opened for `branch`, and PERFORCE will track the integration history between the two files. Subsequent merges of the two files will treat this donor revision as *base*.
- *The target file exists, and was originally branched from the donor file with* `p4 integrate`. In this case, a three-way merge is scheduled between the target and the donor.
- *The target file exists, but was not branched from the donor*. Since these two file revisions did not begin their lives at a common, older file revision, there can be no ***`base`*** file, so `p4 resolve` can't do a three-way merge. In this case, `p4 resolve` will do a *two-way merge*, in which ***`yours`*** is also used as ***`base`***. In a two-way merge, all changes appear as ***`theirs`***, and there can be no conflicts.

Deleting Branches

To delete a branch, use

```
p4 branch -d branchname
```

Deleting a branch deletes only the branch view description, making the branch inaccessible from any subsequent `p4 integrate` commands. If the files in the branched codeline are to be removed, they must be deleted with `p4 delete`.

Advanced Integration Functions

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

Integrating Specific File Revisions

By default, the `integrate` command will integrate into the target all the revisions of the donor since the last donor 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 syntax here is a little strange: although the file provided as an argument to `p4 integrate` is the target, the file revision specifier is applied to the donor.

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 of his* `newinit.c` *file in his branched workspace, he types*

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

The target file is given as an argument, but the file revisions are applied to the donor. 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

Once a particular revision of a donor file has been integrated into a particular target, that particular revision is usually skipped in subsequent integrations with the same target. If all the revisions of a donor have been integrated into a particular target, `p4 integrate` will give the error message `All revisions already integrated`. But integration of a particular donor can be forced, even though integration has already been performed, by providing the `-f` flag to `p4 integrate`.

In previous versions of PERFORCE, re-resolution of files was accomplished with `p4 reresolve`*, which has now been replaced by* `p4 resolve -f`.

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

How Integrate Works

The preceding material in this chapter was written from a user's perspective. This section makes another pass at the same material, this time describing the mechanism behind the integration process.

p4 integrate's Definitions of yours, theirs, *and* base

`yours`, `theirs`, *and* `base` *are first discussed on page 42.*

The values of *`yours`*, *`theirs`*, and *`base`* in a three-way merge are quite different when propagating changes between two codelines:

`yours`	The file that changes are being propagated to (also known as the `target` file). This file is in the client workspace, and it is overwritten by the result once the resolve process is complete. In a forward integrate, this is a file in the branched codeline. When the `-r` flag has been provided to `integrate`, this is a file in the original codeline.
`theirs`	The file revision that changes are being read from (also known as the `donor` file). This file revision comes from the depot, and is unchanged by the resolve process. In a forward integrate, this is a file revision from the original codeline. When the `-r` flag has been provided to `integrate`, this is a file in the branched codeline.
`base`	The last integrated revision of the donor file. When a new branch is created and `integrate` is used 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. It applies the branch view to any target files provided on the command line to produce a list of donor/target file pairs. If no files are provided on the command line, a list of all donor/target file pairs is generated. It notes individually each revision of each donor file that is to be integrated.
2. It discards any donor/target pairs for which the donor file revisions have been integrated in previous changes. Each revision of each file that has been integrated is remembered individually, in order to avoid making the user merge changes more than once.
3. It discards any donor/target pairs whose donor file revisions have integrations pending in files that are already opened in the client.
4. All remaining donor/target pairs will be integrated. The target file is opened on the client for the appropriate action (see below), and merging is scheduled.

Integrate's Actions

The integrate command will take one of three actions, depending on particular characteristics of the donor 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 donor file the target file was branched from. This allows three-way merges to be performed between subsequent donor and target revisions with the original donor file revision as *base*.

`integrate`	If both the donor 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 donor and target must be merged with `p4 resolve`.
`delete`	When the target file exists but no corresponding donor 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 donor tree.

When the `-r` flag is not provided to `p4 integrate`, the original codeline provides the donor files, and the branched codeline provides the targets. When the `-r` flag is given, the branched codeline is the donor, and the original files are the targets.

Integration Reporting

The branching-related reporting commands are:

Command	**Function**
`p4 integrate -n` *`[filepatterns]`*	Reports what `integrate` would do without actually doing it. Any of the usual parameters to `integrate` can be specified as well.
`p4 resolve -n` *`[filepatterns]`*	Reports files that have been scheduled for resolve by `p4 integrate`, but that have not yet been resolved. Reports what `p4 resolve` would do without actually doing it. Any of the usual parameters to `resolve` can be provided as well.
`p4 resolved`	Lists those files that have been resolved, but have not yet been submitted.
`p4 integrated` *`[filepatterns]`*	Describes all integrated and submitted files that match the filepattern arguments.
`p4 branches`	Display a list of all branches known to the system.
`p4 filelog` *`[filepatterns]`*	Describes the revision history of the named files. For each revision of the named files, the following is reported: • change number; • operation (edit, add, delete, branch, integrate) • client name; • user name; and • description. If the operation was `branch` or `integrate`, the names and revisions of the corresponding branch files are reported as well.

There is an ordering to the first four of these commands: the first is performed before `integrate`; the second is done after `integrate` and before `resolve`; the third is given after `resolve` but before `submit`, and the fourth is performed after `submit`. The fifth command provides a complete history of any file, and is incredibly useful.

CHAPTER 10

Job Tracking

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

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

Jobs perform no functions internally to PERFORCE; rather, they are provided as a method of keeping track of 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. Since jobs do nothing more than provide this information to the user, the job reporting facilities are particularly important.

Daemons are described in Chapter 11.

The job facilities in PERFORCE do not provide a full-scale job tracking system. They can be used as is, or integrated with another system via a daemon.

Creating and Editing Jobs

Jobs are created with the `p4 job` command.

Example: Creating a Job

Sarah, who shares the same PERFORCE server as Ed, has found a bug in Elm's filtering code. Ed is fixing code, so Sarah creates a new job with `p4 job` *and fills in the resulting form as follows:*

```
Job:   new

User:   edk

Status: new

Description:
       Filters on the "Reply-To:" field
       don't work.
```

She has changed `User:` *from her username to* `edk`*. Ed will see this job the next time he views any pending changelist with* `p4 submit` *or* `p4 change`*.*

The `p4 job` form's fields are:

Field Name	**Description**	**Default**
`Job`	The name of the job. Whitespace is not allowed in the name.	`new`
`User`	The user whom the job is assigned to, usually the username of the person assigned to fix this particular problem.	PERFORCE user-name of the per-son creating the job.
`Status`	`open`, `closed`, `suspended`, or `new`. An `open` job is one that has been created but has not yet been fixed. A `closed` job is one that has been completed. A `suspended` job is an open job that is not currently being worked on. `New` jobs exist only while the change creation form is open.	`new`; changes to `open` after job creation form is closed.
`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`, or is blank, 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 owner and description can be changed arbitrarily; the status can be changed to any of the three valid status values `open`, `closed`, or `suspended`. If `p4 job` *`jobname`* is called with a non-existing jobname, a new job is created.

Linking Jobs to Changelists, and Changing a Job's Status

Automatically Performed Functions

By default, all open jobs owned by a particular user will appear in all PERFORCE changelists subsequently created by that user. A job is automatically closed when one of its associated changelists is successfully submitted. Jobs can be disassociated from changelists by deleting the job from the changelist's change form, and any job of any status may be added to a changelist.

Example: Including and excluding jobs from changelists

Ed is unaware of the job that Sarah has assigned to him. He is currently working on an unrelated problem; he types `p4 submit` and sees the following:

```
Change: new
Client: edk
User:   edk
Status: new
Description:
        Updating “File” files
Jobs:
        job000125              # Filters on the “Reply-To”
field d

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

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

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

```
Change: new
Client: edk
User:   rlo
Status: new
Description:
        Fixes a number of filter problems
Jobs:
       job000125               # Filters on the “Reply-To” field d

Files:  //depot/filter/actions.c   # edit
        //depot/filter/audit.c     # edit
        //depot/filter/filter.c    # edit
```

Since the job is fixed in this changelist, Ed leaves the job on the form. When he quits from the editor, the job is marked as closed, and will not appear in any subsequent changelists unless it is reopened.

Controlling Which Jobs Appear in Changelists

The types of jobs that appear in new changelists created by a particular user can be controlled through the `p4 user` form. This form's `JobView` field allows one of three values:

Value of `JobView` field	**Description**
`Mine`	When a new changelist is created, automatically include all open jobs owned by the invoking user in the changelist form. This setting of `JobView` is the default.
`None`	Don't include any jobs on new changelist forms.
`All`	Include all open jobs owned by all users in all new changelists forms.

In all three cases, any unwanted job may be deleted from the form before leaving the editor, and additional jobs can be added.

Manually Associating Jobs with Changelists

`p4 fix -c` *`changenum jobname`* can be used to link any job, whether `open`, `closed`, or `suspended`, to any PERFORCE changelist, whether pending or submitted. If the job is open but the changelist has already been submitted, the job is closed. If the job has been closed but the changelist is pending, the job is reopened. Otherwise, the job keeps its current status.

Example: Using `p4 fix` *to attach a job to a changelist*

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

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

Arbitrarily Changing a Job's Status

We've already seen two ways of changing a job's status:

1. A job is automatically closed when an associated changelist is submitted.
2. `p4 fix` will change the status of an open job to `closed` if the associated changelist has already been submitted, and will change the status of a closed job to `open` when the job is linked to a pending changelist.

The status of any job can also be changed by bringing up the job definition form with `p4 job` *`jobname`*, and then changing the status to one of the three allowed values. This is the *only* way of changing a job's status to `suspended`.

Deleting Jobs

A job that has been linked to a changelist can be unlinked from that changelist with `p4 fix -d -c` *`changenum jobname`*. Jobs can be deleted entirely with `p4 job -d` *`job-name`*. .

Integrating to External Defect Tracking Systems

The PERFORCE job reporting functionality can be used to integrate jobs into external programs, such as defect tracking systems, via daemons. Please see the next chapter for more information on daemons.

Job Reporting

The commands that generate reports on jobs are:

Command	**Description**

`p4 jobs`
`[ -l ]`
`[ -s statusval]`
`[ file...]`
	Generates a report of all jobs on the server. Prints the job name, date modified, owner, status, and the first 32 characters of the description for each job.
	`p4 jobs -l` outputs the full description of each job.
	`p4 jobs -s` *`statusval`* can be used to limit the report to only those jobs with a particular status value.
	If any file names are provided on the command line with `p4 jobs` *`files`*, the report will be limited to jobs linked to those changelists that affected files that match the filepatterns.

`p4 fixes`
`[ -j` *`jobName`* `]`
`[ -c` *`change#`* `]`
`[` *`file`* `... ]`
	Lists each job along with the change numbers they've been linked to.
	`p4 fixes -j` *`jobname`* provides information only for changelists linked to that particular job.
	`p4 fixes -c` *`changenum`* lists only those jobs associated with the given change number.
	Any file arguments that are provided will limit the listing to changelists that affect those files.

CHAPTER 11 Change Review & Other Daemons

PERFORCE's change review functionality allows users to be notified via email whenever files that they're interested in have been updated in the depot. Since change review is provided via a background PERL script, or *daemon*, this functionality can be modified. Other daemons can be created to perform entirely different functions.

The primary focus of this chapter is on using the change review daemon; a secondary thread discusses how to create your own daemons. This "daemon creation" topic discusses how a daemon might be created that integrates the PERFORCE job tracking facilities to an external defect tracking system.

Providing Change Review Parameters

Any user wishing to receive email notification of changed files must provide two pieces of information to the PERFORCE server: her email address, and a list of the files and directories that she wants to track (or *subscribe* to). This information is provided via the `p4 user` form.

Example: Providing change review parameters with `p4 user`

Sarah wants to be notified whenever any of the Elm `README` *or document files are changed. She types* `p4 user` *and fills in the resulting form with her email address and file review list:*

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

Once the review daemon is running, she'll be notified by email whenever any of the files in her `Review:` *list are changed. This includes all* `README` *files, and all files in the* `doc` *subdirectory.*

Notification is sent whenever a changelist is submitted by any user that edits, deletes, or adds files that match the `p4 user` form `Reviews:` specification.

Running the Daemon

Change review is implemented via a PERL script, `perfreview.perl`, which can be downloaded from

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

This review daemon must be run under PERL 4 or higher; SENDMAIL is also required. It's usually run from the P4D server account, in the same directory as the P4D server, although it can be installed anywhere. Once it's been installed, do the following:

For more information on `P4PORT`*, see Chapter 11 or Appendix A.*

1. Edit the script and follow the instructions at the top. You may need to change the values of certain values in the script, including the locations of the PERL and SENDMAIL executables.
2. Make sure that the environment variable `P4PORT` is set the proper port so the review daemon can communicate with the p4 server.
3. Run `perfreview.perl` in the background, with

   ```
   perfreview.perl &
   ```

This protection level is described in Chapter 14.

4. Make sure that the review daemon is running as a user with review or superuser access. If protections are in their default non-enabled state, the review daemon will automatically have review access.

The script can be modified to provide any desired functionality. For example, you may want to change the text of the outgoing message, or you might change the script to write change reviews to files instead of sending email.

How the Review Daemon Works

The review daemon is quite simple. Every time a changelist is submitted to the PERFORCE depot, email is sent to every user who has subscribed to review any files that are contained in that changelist. (Unlike other job review systems that you may be familiar with, the PERFORCE system doesn't require the email recipient to respond; the email message is sent for notification purposes only).

The daemon is implemented as follows:

1. The PERFORCE depot is polled for submitted, unreviewed changelists with the command `p4 review`;
2. `p4 reviews` is used to generate a list of reviewers for each of these changelists;
3. SENDMAIL is used to email the `p4 describe` changelist description to each reviewer;
4. The first three steps are repeated every sixty seconds; this value can be changed when the review daemon is installed.

The commands used in steps 2 and 3, `p4 reviews` and `p4 describe`, are straightforward reporting commands. The `p4 review` command used in step 1 is rather unusual. All these commands are described below.

Tracking Reviewed Changelists with Review Counters

The review daemon needs to keep track of which changelists have already had their descriptions emailed to reviewers; in general, most PERFORCE daemons will need to keep track of which changelists have already been processed. The `p4 review` command exists to facilitate this process; it was written solely to support daemon creation and is not likely to be run by an end user.

Every line of output produced by `p4 review` has the following form:

```
Change changenum user <email_addr> (Full_Name)
```

For example:

```
Change 6 edk <edk@elmco.com> (Ed Kuepper)
```

When used with no parameters, `p4 review` will output one line for every changelist that has ever been submitted. With the addition of *review counter* arguments, `p4 review` can limit its output to only those changelists not already reported. A review counter is simply a named counter; each review counter can separately track which changelists have and haven't yet been reviewed.

Review counters are used in two variants of `p4 review`:

`p4 review -t countername -c changenum`	Tells `p4 review` that all changelists between 1 and *`changenum`* have already been reviewed by the review counter named *`countername`*.
`p4 review -t countername`	Reports only those changelists not already reported by review counter *`countername`*.

The review counter names `journal`*,* `job`*, and* `change` *are used internally by PERFORCE;* **use of any of these three names as review counters could corrupt the PERFORCE database!**

Review counters work by storing one counter per review counter in the PERFORCE database. The first variant of `p4 review` above sets a counter value for a particular review counter; the second variant gets the change information for all changelists numbered above that counter. The review daemon's review counter is called `review`; the daemon uses `p4 review -t review` to get a list of all unreviewed changelists, and uses `p4 review -t review -c` *`lastchangenum`* to store the highest numbered reviewed change number in the database under the name `review`.

PERFORCE can store any number of review counters; each counter is identified by a unique name, which can be up to ten characters long and cannot contain whitespace.

The `perfreview.perl` daemon's review counter is named `review`; if you're using `perfreview.perl` and are writing your own daemon, don't name the review counter `review`.

Change Review and Protections

Access levels are covered in Chapter 14.

The change review daemon runs at the access level of the user who invokes it (typically `root`), *not* at the access level of the user receiving email. Thus, if the daemon is edited to perform additional functionality, it should not mail out the contents of files unless it is safe for *all* users to see those files, since a user needs only `list` access in order to invoke the `p4 user` command to subscribe to particular files.

Creating Other Daemons

To create another daemon, use `perfreview.perl` as a starting point and change it as needed. One such daemon might upload PERFORCE job information into an external bug tracking system after changelist submission. It would use the `p4 review` command with a new review counter to list new changelists, and use `p4 fixes` to get the list of jobs fixed by the newly submitted changelists. This information might then be fed to the external system, notifying it that certain jobs have been completed.

If you do write a daemon of your own, and would like to share it with other users, please let us know about it at `info@perforce.com`.

Change Review Reporting

The change review daemon uses two reporting commands that have not yet been discussed; an additional reporting command describes the counters tracked by the P4D server:

Command	Meaning
`p4 reviews [ -c changenum ] [ files... ]`	Provides a list of all users who have subscribed to review any files. To see a list of reviewers for the files affected by a particular changelist, use `p4 reviews -c` *`changenum`*. `p4 reviews` *`files`* can also be used to report the reviewers of only those files that are provided as arguments.
`p4 users`	Describe the users currently known to the P4D server. The report includes user names, their email addresses, their full names, and the last time they logged in.
`p4 counters`	Provides a list of counters known to the current P4D server, along with their values.

CHAPTER 12

Reporting and Data Mining

PERFORCE's reporting commands supply information on all data stored within the depot. There are many reporting commands; in fact, there are almost as many reporting commands as there are action commands. These commands have been discussed throughout the 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; discussion of all options for each command is beyond the scope of this manual. For a full description of any particular command, please consult the *PERFORCE Command Reference*, or type `p4 help` *`command`* at the command line.

One previously mentioned note on syntax is worth repeating here: any *filespec* argument in PERFORCE commands, as in

```
p4 files filespec
```

will match any file pattern that is supplied in local syntax, depot syntax, or client syntax, with any PERFORCE wildcards. Brackets around *filespec* means that the file specification is optional.

Files

The commands that report on files fall into two categories: those that give information about file *contents*, (e.g. `p4 print`, `p4 diff`), and those that supply information on file *metadata*, the data that describe a file with no reference to content (e.g. `p4 files`, `p4 filelog`). The first set of reporting commands discussed in this section describes file metadata; the second set describes file contents.

File Metadata

Basic File Information

To view information about single revisions of one or more files, use `p4 files`. This command provides the locations of the files within the depot, the actions (`add`, `edit`, `delete`, etc.) 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)
```

`p4 files` requires one or more filespec arguments. Filespec arguments can, as always, be provided in PERFORCE or local syntax, but the output will always report the corresponding files within the depot. If no revision number is provided, the head revision will be used.

Unlike most other commands, `p4 files` will describe deleted revisions, instead of 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 your 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#revison#`*
...all files at change *n*, whether or not the file was actually included in change n	`p4 files @`*`n`*
...a particular file within a particular label	`p4 files `*`filespec@labelname`*

File Revision History

The revision history of a file is provided by `p4 filelog`. One or more file arguments must be provided, and since the point of `p4 filelog` is to list information about each revision of particular files, file arguments to this command may not contain a revision specification.

The output of `p4 filelog` has this form:

```
   ... #3 change 23 edit on 1997/09/26 by edk@doc ’Updated hel’
... #2 change 9 edit on 1997/09/24 by lisag@src ’Made chang’
... #1 change 3 add on 1997/09/24 by edk@doc 'Added the f’
```

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, 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.>
```

To See File Revision Information...	**Use This Command:**
...including revisions of specific files, with a short description of each changelist the file was submitted in	`p4 filelog `*`filespec`*
...with the full description of each changelist	`p4 filelog -l `*`filespec`*

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 filepattern argument, `p4 opened` will print 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`
...whether or not a specific file is opened by you	`p4 opened` *`filespec`*
...whether or not a specific file is opened by anyone	`p4 opened -a` *`filespec`*

Relationships Between Client and Depot Files

It is often useful to know how the client and depot are related at a particular moment in time. Perhaps you simply want to know where a particular client file is mapped to within the depot, or you may want to know whether or not the head revision of a particular depot file has been copied to the client. The commands that express the relationship between client and depot files are `p4 where`, `p4 have`, and `p4 sync -n`. The first of these commands, `p4 where,` is used to determine where client files would be mapped through the client view into the depot, and vice-versa. `p4 have` tells you which depot files and revisions are available within your client workspace, and `p4 sync -n` describes which files would be read into your client workspace the next time you perform a `p4 sync`.

All these commands can be used with or without filespec arguments. `p4 sync -n` is the only command in this set that allows revision specifications on the filespec arguments.

The output of `p4 where` looks like this:

```
//depot/elm_proj/doc/Ref.guide: //edk/doc/Ref.guide
```

`p4 have`'s output has this form:

```
//depot/doc/Ref.guide#3 - /u1/rlo/edk/elm/doc/Ref.guide
```

and `p4 sync -n` provides output like:

```
//depot/doc/Ref.guide#3 - updating /usr/edk/elm/doc/Ref.guide
```

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 in the client workspace maps to in the depot	`p4 where` *`filespec`*

To See...	Use This Command:
...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

The contents of any file revision within the depot can be viewed 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; The file will be printed at the specified revision; if no revision is specified, the head revision will be 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`*

File Content Comparisons

A client workspace file can be compared to any revision of the same file within in the depot with `p4 diff`. This command takes a filespec argument; if no revision specification is supplied, the workspace file is compared against the revision last read into the workspace.

The `p4 diff` command has many options available; only a few are described in the table below. For more details, please consult the *PERFORCE Command Reference*.

Whereas `p4 diff` compares a client workspace file against depot file revisions, `p4 diff2` can be used to compare *any* two revisions of a file. It can even be used to compare revisions of different files. `p4 diff2` takes two file arguments; wildcards are allowed, but any wildcards in the first file argument must be matched with a corresponding wildcard in the second. This makes it possible to compare entire trees of files.

There are many more flags to `p4 diff` then are 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`*

To See the Differences between...	Use This Command:
...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`*
...all files within two branched codelines	`p4 diff2 //depot/codeline1/... //depot/codeline2/...`
...a file within the client workspace and the revision last taken into the workspace, passing the *context diff* flag to the underlying diff.	`p4 diff -dc` *`file`*

The last example above bears further explanation; to understand how this works, it is necessary to discuss how PERFORCE implements and calls underlying *diff* routines.

PERFORCE uses two separate *diff*s: one is built into the P4D server, and the other is used by the P4 client. Both *diff*s contain identical, proprietary code, but are used by separate sets of commands. The client *diff* is used by `p4 diff` and `p4 resolve`, and the server *diff* is used by `p4 describe`, `p4 diff2`, and `p4 submit`.

PERFORCE's built-in *diff* routine allows three `-d`<*flag*> flags: `-du`, `-dc`, and `-dn`; both `p4 diff` and `p4 diff2` allow any of these flags to be specified. These flags behave identically to the corresponding flags in the standard UNIX *diff*.

Although the server must always use PERFORCE's internal *diff* routine, the client *diff* can be set to any external *diff* program by pointing the `P4DIFF` environment variable to the full path name of the desired executable. Any flags used by the external *diff* can be passed to it with `p4 diff`'s `-d` flag. Flags are passed to the underlying *diff* according to the following rules:

- If the character immediately following the `-d` is not a single quote, then all the characters between the `-d` and whitespace are prepended with a dash and sent to the underlying *diff*;
- If the character immediately following the `-d` is a single quote, then all the characters between the opening quote and the closing quote are prepended with a dash and sent to the underlying *diff*.

The following examples demonstrate the use of these rules in practice.

If you want to pass the following flag to an external client *diff* program:	Then call `p4 diff` this way:
`-u`	`p4 diff -du`
`--brief`	`p4 diff -d-brief`
`-C 25`	`p4 diff -d'C 25'`

Changelists

Two separate commands are used to describe changelists. The first, `p4 changes`, lists changelists that meet particular criteria, without describing the files or jobs that make up the changelist. The second command, `p4 describe`, lists the files and jobs affected by a single changelist. These commands are described below.

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 the following:

```
   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, or those affecting a particular file, or the last *n* changelists. Currently, the output can’t be restricted to changelists submitted by particular users, although simple shell or Perl scripts can be written to implement this.

A Korn shell script that implements this report is described in “Reporting with Scripting” on page 94.

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`*
...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`*
...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`*

Very few PERFORCE commands allow revision ranges to be appended to file specifications. For details on revision ranges, please see page 36.

Files and Jobs Affected by Changelists

Additional commands that report on jobs and changelists are described in the job reporting section of this chapter (page 91).

To view a list of files and jobs affected by a particular changelist, along with the *diff*s of the new file revisions and the previous revisions, use `p4 describe`. Its output 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
    < The second way, used most commonly when transmitting
    ---
    > The second way, which is commonly used when transmitting

...<etc.>
```

This output is quite lengthy; a shortened form that eliminates the file *diff*s can be generated with `p4 describe -s` *`changenum`*.

To See:	Use This Command:
...a list of all files submitted and jobs fixed by a particular changelist, displaying the *diff*s 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 *diff*s	`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 @`*`changenum filespec`*

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 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`*[a]
...what a particular `p4 integrate` variation would do, without actually doing the integrate.	`p4 integrate [`*`args`*`] -n [`*`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`*`]`

To See:	Use This Command:
...a list of integrated, submitted files that match the filespec arguments	`p4 integrated` *`filespec`*
...a description of the revision history of the named files, including the following for each file revision: • change number; • operation (`add`, `edit`, `delete`, `branch`, or `integrate`) • client name; • user name; and • description.	`p4 filelog [`*`filespec`*`]`

a. In this case, the filespec should be presented in depot syntax, and should represent the branched codeline. For example, if a codeline had been branched to `//depot/r22/...`, then a list of all files in the branched codeline would be obtained with `p4 files //depot/r22/...`

Job Reporting

Job reporting is accomplished with two commands. The first, `p4 jobs`, reports on all jobs known to the system; the second command, `p4 fixes`, reports only on those jobs that have been attached to changelists. Both these commands have numerous options.

Basic Job Information

To see a list of all jobs known to the system, use `p4 jobs`. Options to this command can be used to specify criteria for describing only particular types of jobs; for example, the `-s` flag will limit the report to jobs of a particular status. `p4 jobs` 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 'Can’t read filters w'
```

Its output includes the jobs name, date entered, job owner, and the first 32 characters of the job description. The job status is included if the job is open or suspended; closed jobs are indicated by the absence of job status from the report.

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`
...of a particular status (`open`, `closed`, or `suspended`)	`p4 jobs -s` *`status`*

To See a List of Jobs:	Use This Command:
...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 fix` looks like this:

```
   job000302 fixed by change 634 on 1997/09/01 by edk@eds_mach
filter_bug fixed by change 540 on 1997/10/22 by edk@eds_mach
```

A number of options allow the reporting of only those changes that fix a particular job, or jobs fixed by a particular changelist, or jobs fixed by changelists that are linked to particular files.

A fixed job is not the same as a closed job, since `open` jobs can be linked to pending changelists, and `pending` jobs can be reopened even after the associated changelist has been submitted. To list jobs with a particular status, use `p4 jobs`.

To See a Listing of...	Use This Command:
...all fixes for all jobs	`p4 fixes`
...all changelists linked to a particular job	`p4 fixes -j` *`jobname`*
...all jobs linked to a particular changelist	`p4 fixes -c` *`changenum`*
...all jobs fixed by changelists that contain particular files	`p4 fixes` *`filespec`*
...all jobs fixed by changelists that contain particular files, including changelists that contain files that were later integrated with the specified files	`p4 fixes -i` *`filespec`*

Reporting for Daemons

The PERFORCE change review mechanism uses the following reporting commands. Any of these commands might also be used with user-created daemons. For further information on daemons, please see chapter 11, and consult the source code of the PERFORCE Change Review daemon.

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`*

To list...	**Use This Command:**
...all users who have subscribed to review particular files	`p4 reviews` *`filespec`*
...all users who have subscribed to read any files in a particular changelist	`p4 reviews -c` *`changenum`*
...a particular user's email address	`p4 users` *`username`*

System Configuration

Three commands report on the PERFORCE system configuration. One command reports on all PERFORCE users; another prints data describing all PERFORCE client workspaces, and a third reports on PERFORCE depots.

`p4 users` generates its data as follows:

```
   edk <edk@eds_ws> (Ed Kuepper) accessed 1997/07/13
lisag <lisa@lisas_ws> (Lisa Germano) 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.

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.

The `-o` 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 will simply tell you what they would ordinarily do.

The `-n` flag can be used with the following commands:

```
p4 integrate       p4 resolve
p4 labelsync       p4 sync
```

Reporting with Scripting

Although PERFORCE's reporting commands are sufficient for most needs, there may be times when you want to view data in a format that PERFORCE doesn't directly support. In these situations, the reporting commands can be used in combination with scripts to print only the data that you want to see. Three examples are provided here.

Comparing the Change Content of Two File Sets

To compare the "change content" of two sets of files, it is necessary to *diff* them externally. To do this, run `p4 changes` twice, once on each set of files, and then use any external *diff* routine to compare them.

In the following example, `main` represents the main codeline, and `r98.4` is a codeline that was originally branched from `main`:

```
    p4 changes //depot/main/... > changes-in-main
p4 changes //depot/r98.4/... > changes-in-r98.4
diff changes-in-main changes-in-r98.4
```

This could be used to uncover which changes have been made to `r98.4` that haven't been integrated back into `main`.

Changelists Submitted by Particular Users

The `p4 changes` command does not have a flag that allows only those changes submitted by particular users to be viewed, but this can be accomplished by grepping the output of `p4 changes`. For example, in the Korn shell, create an executable file with these contents:

```
p4 changes | grep '.* by '$1'@
```

When this script is called with a username as an argument, only those changes created by that user will be printed.

Listing Subdirectories of the Depot

Although all files in the depot can be listed with `p4 files`, there is no option for reporting only the names of subdirectories within the depot. However, this can be accomplished with the following Perl script, which takes file arguments in either PERFORCE or local syntax:

```
   open(P4,"p4 files " . join(' ',@ARGV) . " |");
while(<P4>) {
    s@^(.*)/[^/]*#.*$@$1@;
    if(!exists $d{$_}) {
    print;
    $d{$_}=42;
  }
}
```

These scripting examples are, of course, non-exhaustive. Use scripts whenever you want to generate reports that can't be created through existing PERFORCE commands.

CHAPTER 13

System Administration: Installation and Maintenance

Installation of P4D and P4

NT/95

Installation of PERFORCE on Windows machines is handled by the installer. If you're using PERFORCE on Windows, you can skip to the protections and journaling sections, both of which start on page 98.

PERFORCE operation requires two executables: P4D, the server, and P4, the client. If you haven't already downloaded these, they may be retrieved from `http://www.perforce.com/perforce/load.html`.

The P4 program typically resides in `/usr/local/bin`, and P4D is usually located in `/usr/local/bin` or in its own root directory (see below), although they can be installed anywhere. The P4 program can be installed on any server that has TCP/IP access to the P4D host. To limit access to the P4D server files, it is recommended that P4D be owned and run by a PERFORCE user account.

Only a few steps need to be performed before P4 and P4D can be run: a root directory for the P4D files is created, a TCP/IP port is provided to P4`d`, and P4 is provided the name of the P4D host and number of the P4D port. These steps are described in the following sections.

Creating a P4D Root Directory

P4D stores all of its data in files and subdirectories of its own root directory, which can reside anywhere on the server system. This directory can be named anything at all, and the only necessary permissions are read and write for the user who starts P4D. Since P4D will store all submitted files in this directory, the size of the directory can eventually become quite large. Disk space management is described on page 100.

The environment variable `P4ROOT` should be set to point to this directory. Alternatively, the `-r` flag can be provided when P4D is started (see below). The P4 clients never directly use this directory, so they don't need to know the value of `P4ROOT`.

Setting P4D's Port

P4D and P4 communicate via TCP/IP. When P4D starts, it will, by default, listen on port 1666. The P4 client will, by default, assume that its P4D server is located on host `perforce`, listening on port 1666.

If P4D is to listen on a different port, the port can be specified with the `-p` flag when starting P4D (example: `p4d -p 1818`), or the port can be set with the `P4PORT` environment variable. Chances are that your P4D host is not named `perforce`, but you can simplify life somewhat for your P4 users by setting `perforce` as an alias to the true host name in the host's `/etc/hosts` file, or by doing so via Sun's NIS or Internet DNS.

Telling P4 Where The P4D Server Is

The P4 client program needs to know on which TCP/IP port the P4D server program is listening. P4 can be told which host and port the P4D server program is listening on by setting each P4 user's `P4PORT` environment variable to *host:port#* , where *host* is the name of the host that P4D is running on, and *port#* is the port that P4D is listening on.

Examples:

If P4PORT is...	Then...
`dogs:3435`	P4 will communicate with the P4D server on host dogs listening at port 3435.
`x.com:1818`	P4 will communicate with the P4D server on host x.com listening on port 1818.

The definition of `P4PORT` can be shortened if P4 is running on the same host as P4D. In this case, only the P4D port number need be provided to P4. If P4D is running on a host named or aliased `perforce`, listening on port `1666`, the definition of `P4PORT` for the P4 client can be dispensed with altogether.

- Examples:

If P4PORT is...	Then...
`3435`	P4 will communicate with the P4D server on its local host listening at port 3435.
`<not set>`	P4 will communicate with the P4D server on the host named or aliased `perforce` listening on port 1666.

Starting P4D: The Basics

After P4D's `P4PORT` and `P4ROOT` environment variables have been set, P4D can be run in the background with the command

```
p4d &
```

`P4PORT` can be overridden by starting P4D with the `-p` flag, and `P4ROOT` will be ignored if the `-r` flag is provided. The startup command would then have this form:

```
p4d -r /usr/local/p4files -p 1818 &
```

Although this command is sufficient to run P4D, other flags, which control such things as error logging, checkpointing, and journaling, can and should be provided. These flags are discussed in the next two sections.

Logging Errors

The P4D program tries to ensure that all error messages reach the user, but if an error occurs and the client program disconnects before the error is sent, P4D logs the error to its error output. The error output file can be specified with the `-L` flag to P4D, or can be defined in the environment variable `P4LOG`. If no error output file is defined, errors are dumped to the P4D program's standard error.

Protections

By default, every PERFORCE user is a PERFORCE superuser, and can run any PERFORCE command on any file. These default access levels are changed easily with the `p4 protect` command; the administrator who installs PERFORCE should run `p4 protect` soon after installing P4 and P4D. Access levels are described in detail in the next chapter.

Checkpointing, Journaling, and Recovery

Checkpointing and journaling archive only the database files, not *the files in the depot directories! Back up the depot files with the standard OS backup commands* after *checkpointing; this is because the static checkpoint can be dumped and restored consistently, while the dynamic database files may change during the course of backing up and may be inconsistent.*

Two sets of files in the P4D root directory are sufficient to recover the complete P4D server data:

- P4D stores its metadata in the top level of its root directory; all of these files are binary, and begin with "`db.`".
- Files submitted by P4 clients are stored in P4D subdirectories that have the same names as the depots. By default, there is only one of these subdirectories; its name is `depot`.

P4D provides checkpointing and journaling facilities for recovery of the metadata database files. A checkpoint is just a snapshot or copy of the database files at a particular moment in time, and a journal is a log that records updates since that snapshot. If the database files become corrupted, or if there's a disk crash, you can recover the files by reloading the checkpoint and applying the changes stored in the journal. Both the checkpoint and journal are text files, and have the same format.

Because the data stored in the PERFORCE database is irreplaceable, journaling and checkpointing should be performed early and regularly.

Making a Checkpoint

You can create a checkpoint by invoking the P4D program with the `-jc` flag:

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

This can be run while the P4D server is running. To make the checkpoint, P4D locks the entire database and then dumps the database contents to a file named `checkpoint.`*x*, where *x* is a sequence number. Before unlocking the database, P4D copies and then truncates the journal file if it exists. This action guarantees that the last checkpoint combined with the current journal always reflects the database.

A different checkpoint file can be specified by typing a filename after the `-jc` flag:

```
% p4d -r root -jc /disk2/perfback/ckp
```

A checkpoint file may be compressed, archived, or moved onto another disk. At that time or shortly thereafter, the files in the depot subdirectories should be archived as well. When recovering, *the checkpoint must be no newer than the files in the depots*.

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

Since checkpoints are readable only by the server version that created them, a new checkpoint should be taken every time the P4D server is upgraded.

Regular checkpointing is important if journaling is turned on, to keep the journal from getting too long. Making a checkpoint just before dumping the file system is simply good practice.

Turning Journaling On and Off

By default, the journal is written to the file journal in the P4D root directory. The name and location of this file can be changed by specifying the name of the journal file in the environment variable `P4JOURNAL`, or by providing a `-J` filename flag to P4D. In either case, the journal file name can be provided as an absolute path, or as a path relative to the P4D server root.

To create a journal, do one of the following:

- Create an empty journal file in the server root, then start P4D;
- Set the `P4JOURNAL` environment variable to point to any file, then start P4D; or
- Start P4D with the `-J` filename flags.

Be sure to create a new checkpoint with the `p4d -jc` flag immediately after taking a checkpoint, since a journal without a checkpoint is useless. Since there is no sure protection against disk crashes, the journal file and the P4D root directory should be located on different disks.

To disable journaling, remove the existing default journal file (if it exists), unset the environment variable `P4JOURNAL`, and type the P4D command again without the `-J` flag.

Recovering

If the database files become corrupted or lost, either because of disk errors, errors in the P4D program itself, or a disk crash, the files can be recreated with your stored checkpoint and journal. To do this, kill P4D, remove the database (“`db.`”) files, and then invoke the P4D program with the `-jr` flag:

```
 % rm root/db.*
% p4d -jr checkpoint_file journal_file
```

After recovering the database you will need to restore the depot files with the UNIX restore command to ensure that they are as new as the database.

Managing Disk Space

All of P4D's stored source files reside in subdirectories of the P4D root, as do its database files, and by default the checkpoints and journals. The stored source file depots are grow-only, and this can clearly present disk space problems on high use systems. The following approaches may be used to remedy this:

- Store the journal file on a separate disk, using the `P4JOURNAL` environment variable or the `-J` flag to `p4d`.
- Checkpoint on a daily basis to keep the journal file short.
- Compress checkpoints after they are made.
- Use the `-jc` *checkpointfile* option with the P4D command to write the checkpoint to a different disk. Or use the default checkpoint files, but backup your checkpoints and then delete them from the root directory. They aren't needed unless you are recovering.
- **[UNIX only]** If necessary, all or part of the depot subdirectories may be relocated to other disks, using symbolic links from the depot trees. This should be done while the P4D server is not running.
- The database files can become internally bloated, due to the nature of the implementation of their access methods. They can sometimes be compressed by recreating them from a checkpoint: take a checkpoint, delete the database files, then recover. This should only be done if the database files are more than about 10 times the size of the checkpoint in total.

License File

PERFORCE servers are licensed according to how many users they will support. This licensing information lives in a file called license in the P4D root directory. It is a plain text file supplied by PERFORCE Software. Without this license file, the P4D server will limit itself to two users and two client workspaces.

The current licensing information can be viewed by invoking `p4d -V` from the server root directory.

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

Release and License Information

The P4 client and P4D server programs will display their version and license information with the `-V` flag. The `p4 info` command will attempt to connect to the server and display its license information (among other things). Version information can also be gleaned from P4 or `p4d` executables with the UNIX `what(1)` command.

CHAPTER 14

System Administration: Protections

PERFORCE provides a protection scheme to prevent unauthorized or inadvertent access to the depot. The protections determines which PERFORCE commands can be run, on which files, by whom, and from which host. Since any user can change their PERFORCE username with `P4USER`, user level protections provide safety, not security. At the host level, protections are as secure as the host itself.

Protections are set with the `p4 protect` command.

When Should Protections Be Set?

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

The PERFORCE protections are stored in the `db.protect` file in the server root directory; if `p4 protect` is first run by an unauthorized user, the depot can be brought back to its unprotected state by removing this file.

Setting Protections with p4 protect

The `p4 protect` form contains a single field with multiple lines. Each line specifies a particular permission; the contents look something like this:

Sample protections table

```
Protections:
    read     emily    *             //depot/elm_proj/...
    write    *        195.3.21.*    //...
    write    joe      *             -//...
    write    lisag    *             -//depot/...
    write    lisag    *             //depot/doc/...
    super    edk      *             //...
```

(The four fields may not line up vertically on your screen; they are aligned here for ease of reading).

The Permission Lines' Four Fields

Each line specifies a particular permission; each permission is always described by four fields. The meanings of these four fields are:

Field	Meaning
`Access Level`	Which access level is being granted: `list`, `read,` `open`, `write`, `super`, or `review`. These are described below.
`User`	The user whose protection level is being defined. This field can contain the '`*`' wildcard: '`*`' by itself would grant this protection to everyone; '`*e`' would grant this protection to everyone whose username ends with an '`e`'. Since PERFORCE usernames can be changed by setting `P4USER`, this field provides safety, not security.
`Host`	The TCP/IP address of the host being granted access. This must be provided as the numeric address of the host in dotted quad notation (e.g. `206.14.52.194`). This field may contain the '`*`' wildcard. A '`*`' by itself means that this protection is being granted for all hosts. The wildcard can be used as in any string, so '`127.30.41.*`' would define access to any subnet within `127.30.41`, and '`*3*`' would refer to any IP address with a '`3`' in it. Since the client's IP address is provided by IP itself, this field provides as much security as is provided by the network.
`Files`	The files in the depot that permission is being granted on. '`//...`' means all files in all depots.

Access Levels

The access level is described by the first field; the six access levels are:

Access Level	Meaning
`list`	Permission is granted to run PERFORCE commands that display data *about* files, (e.g. `p4 filelog)`. No permission is granted to view or change the contents of the files.
`read`	The user(s) can run those PERFORCE commands that are needed to read files, such as `p4 client` and `p4 sync`. The `read` permission includes `list` access.
`open`	Grants permission to read files from the depot into the client workspace, and gives permission to open and edit those files. This permission does *not* allow the user to write the files back to the depot. `open` is similar to `write`, except that with `open` permission, users are not allowed to run `p4 submit` or `p4 lock`. The open permission includes `read` and `list` access.
`write`	Permission is granted to run those commands that edit, delete, or add files. `Write` permission includes `read`, `list,` and `open` access. This permission allows use of all PERFORCE commands except `protect`, `depot`, `obliterate`, and `verify`.

Access Level	Meaning
`review`	A special permission granted to review daemons. It includes `list` and `read` access, plus use of the `p4 review` command. It is needed *only* by review daemons.
`super`	For PERFORCE superusers; grants permission to run all PERFORCE commands. Provides `write` and `review` access plus the added ability to edit protections, create depots, obliterate files, and verify files.

Each PERFORCE command is associated with a particular minimum access level; for example, to run `p4 sync` on a particular file, the user must have been granted *at least* `read` access on that file. The access level required to run a particular command can usually be reasoned from knowledge of what the command does; for example, it is somewhat obvious that `p4 print` would require `read` access. A full list of the minimum access levels required to run each PERFORCE command is provided on page 105.

Which Users Should Receive Which Permissions?

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

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

Because `open` access allows local editing of files, but doesn't allow these files to be written to the depot, `open` access is usually granted only in unusual circumstances. Choose `open` access over `write` access when users will be testing their changes locally, but when these changes should not be seen by other users. For example, bug testers may want to change code in order to test theories as to why particular bugs occur, but these changes would be for their own use, and would not be written to the depot. Or, a codeline might be frozen, with local changes submitted to the depot only after careful review by the development team. In this case, `open` access would be granted until the code changes have been approved; at that time, the protection level would be upgraded to `write`.

Default Protections

When `p4 protect` is first run, two permissions are set by default. The default protections form looks like this:

```
Protections:
        write      *      *       //...
        super      edk    *       //...
```

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

Interpreting Multiple Permission Lines

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

Example:
How protections work.

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

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

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

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

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

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

Exclusionary Protections

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

To use exclusionary mappings properly, it is necessary to understand some peculiarities associated with them:

- When an exclusionary protection is included in the protections table, the order of the protections is relevant: the exclusionary protection is used to remove any matching protections *above* it in the table.
- No matter what access level is provided in an exclusionary protection, *all* access levels for the matching files and IP addresses are denied. The access levels provided in exclusionary protections are irrelevant.

The reasons for this seemingly strange behavior are described in the section *How Protections are Implemented* on page 106.

Example: Exclusionary protections

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

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

The second permission seemingly grants `write` *access to all users to all files in all depots, but this is overruled by later exclusionary protections for certain users:*

- *The third permission denies Joe permission to access any file from any host. No subsequent lines grant Joe any further permissions; thus, Joe has been effectively locked out of PERFORCE.*
- *The fourth permission denies Lisa all access to all files on all hosts, but the fifth permission gives her back* `write` *access on all files within a specific directory. If the fourth and fifth lines were switched, Lisa would be unable to run any PERFORCE command*

Access Levels Required by PERFORCE Commands

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

Command	Access Level	Command	Access Level
add	open	jobs [a]	list
branch [a]	open	label [a]	open
branches	list	labels [a]	list
change	open	labelsync	open
change -f	super	lock	write
changes [a]	list	obliterate	super
client	list	open	open
clients [a]	list	opened	list
delete	open	print	read
depot [a]	super	protect [a]	super
depots [a]	list	refresh	read
describe	read	reopen	open
describe -s	list	reresolve	open
diff	read	resolve	open
diff2	read	resolved	open
edit	open	revert	open
files	list	review [a]	review
filelog	list	reviews [a]	list
fix [a]	open	submit	write

Command	Access Level	Command	Access Level
`fixes` [a]	`list`	`sync`	`read`
`have`	`list`	`unlock`	`open`
`help`	`none`	`user` [a]	`list`
`info`	`none`	`users` [a]	`list`
`integrate` [b]	`open`	`verify`	`review`
`integrated`	`list`	`where` [a]	`none`
`job` [a]	`open`		

a. This command doesn't operate on specific files. Thus, permission is granted to run these commands if the user has the specified access to at least one file in the depot.

b. To run `p4 integrate`, the user needs `open` access on the target files and `read` access on the donor files.

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

How Protections are Implemented

This section describes the algorithm that PERFORCE follows to implement its protection scheme. Protections can be used properly without reading this section; the material here is provided to explain some of the more eccentric behavior described above.

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

- The command is looked up in the command access level table shown on page 105 to determine the *minimum* access level needed to run that command. In our example, `p4 print` is the command, and the minimum access level required to run that command is `read`.
- PERFORCE makes the first of two passes through the protections table. Both passes move up the protections table, bottom to top, looking for the first relevant line. The first pass is used to determine whether or not the user is allowed to know whether or not the file exists, and this search simply looks for the first line encountered that matches the user name, host IP address, and file argument. If the first matching line found is an inclusionary protection, then the user has permission to list the file, and PERFORCE proceeds to the second pass. If the first mapping found is an exclusionary mapping , or if the top of the protections table is reached without a matching protection being found, then the user has no permission to even list the file, and will receive a message like `File not on client`.
- As an example, suppose that our protections table is set as follows:

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

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

- If the first pass is successful, a second pass is made at the protections table, again reading bottom to top; this pass is the same as the first, except that access level is now taken into account. If an inclusionary protection line is the first line encountered that matches the user name, IP address, file argument, and has an access level greater than or equal to the access level required by the given command, then the user is given permission to run the command. If an exclusionary mapping is the first line encountered that matches according to the above criteria, or if the top of the protections table is reached without finding a matching protection, then the user has no permission to run the command, and will receive the message `You don’t have permission for this operation`.

CHAPTER 15

System Administration: Superuser Commands

Three PERFORCE commands can be used only by users with PERFORCE superuser privileges. These commands allow the superuser to verify files using 128-bit signatures, remove all traces of file from the depot, create multiple depots on the local server, or provide read access to files on other servers.

File Verification by Signature

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

For more information about the MD5 algorithm, which is used to generate the file signatures, please see `<http://backupvault.com/md5.htm>.`

File Obliteration

The depot is always growing. Obviously, this is not always desirable: a branch might be performed incorrectly, creating hundreds of unneeded files; or perhaps there are simply a lot of old files around that are no longer being used. `p4 delete` won't help, since this command marks the file as deleted in its head revision, but leaves the old revisions intact.

`p4 obliterate filename` can be used by superusers to remove all traces of a file from a depot, making the file indistinguishable from one that never existed in the first place. `p4 obliterate` is so destructive, in fact, that we haven't even told you how it really works yet. `p4 obliterate filename` only reports on what it *will* do; to actually destroy the files, use `p4 obliterate -y filename`.

Changelist Deletion & Description Editing

The `-f` flag can be used with `p4 change` to change the description or username of submitted changelists. The syntax is `p4 change -f` *`changenumber`*; this presents the standard changelist form, in which the description and/or username may be edited.

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

Distributed Depots

PERFORCE *distributed depots* allow the P4 client program to access files from multiple depots. These other depots may reside within the P4D server normally accessed by the P4 client program, or they may reside within other, *remote*, P4D servers.

The P4 client's local P4D server program acts as a proxy client to the remote server programs, so the client doesn't need to know where the files are actually stored, and doesn't need direct access to the remote P4D server programs.

The use of distributed depots on remote servers is currently limited to read-only operations; thus, a P4 client program may not add, edit, delete or integrate files that reside in depots on other servers. Depots sharing the same P4D server as the client are not subject to this limitation.

Defining New Depots

New depots in a server namespace are defined with the command `p4 depot` *`depotname`*. If called with the default depotname `depot`, the `p4 depot` command will bring up the following form:

```
Depot Name:   depot
Type:         local
Address:      subdir
Map:          depot/...
```

When `p4 depot depot` is called, the form is filled in with values representing the state of the default depot. Its name, of course, is `depot`. It resides in the local P4D server namespace; so its type is `local`, (as opposed to `remote`). The `Map`: field indicates where the depot subdirectory is located relative to the root directory of the P4D server program; in this default case, the depot called `depot` starts in the depot subdirectory directly underneath the root.

Defining Local Depots

To define a new local depot (that is, a new depot in the current P4D server program namespace), `p4 depot` is called with the new depot name, and only the `Map:` field in the resulting form need be changed. For example, to create a new depot called `book` with the files stored in the local P4D server namespace in a root subdirectory called `manual`, the command `p4 depot book` would be typed, and the resulting form would be filled in as follows:

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

Defining Remote Depots

Defining a new depot on a remote P4D server is only slightly more complicated. The `Type`: is `remote`; the server address must be provided in the `Address`: field, and the `Map`: field must be given a mapping into the remote depot namespace.

Example: Defining a remote depot

Lisa is working on a GUI for Elm. She and Ed are using different P4D servers; his is on host `pine`*, and it's listening on port* `1818`*. Lisa wants to grab Ed's GUI routines for her own use; she knows that Ed's color routine files are located on his P4D server's single depot under the subdirectory* `graphics/GUI`*. Lisa's first step towards accessing Ed's files would be to create a new depot. She'll call this depot* `gui`*; she'd type* `p4 depot GUI` *and fill in the form as follows:*

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

This creates a remote depot called `gui` *on Lisa's P4D server; this depot maps to Ed's depot's namespace under its* `graphics/gui` *subdirectory.*

The Mapping Field, and What it Means

The `Map`: field is analogous to a client's view, except that the view may contain multiple mappings and the `Map`: field always contains a single mapping. This single mapping format changes depending on whether or not the depot being defined is local or remote:

- If a local depot is being defined, the mapping should contain a subdirectory relative to the file space of the P4D server root directory. For example, `graphics/gui/...` maps to the `graphics/gui` subdirectory of the P4D server root.
- If a remote depot is being defined, the mapping should contain a subdirectory relative to the remote depot namespace. For example, `//depot/graphic/gui/...` would map to the `graphic/gui` subdirectory of the remote server depot named `depot.`

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

Naming Depots

Depot names share the same namespace as branches, clients, and labels. For example, `//foo` refers unambiguously to either the depot `foo`, the client `foo`, the branch `foo`, or the label `foo`.

Accessing Files In Other Depots

Files from any remote or local depot known to the default P4D server can be accessed simply by using the depot's name wherever the default depot name `depot` is usually used. This means that any defined depot name can be used in the following ways:

- As part of any P4 command that takes depot syntax. For example, the following command will retrieve all files in the subdirectory `foo` of depot `bar`:

```
p4 sync //bar/foo/...
```

- On the left-hand side of any client view. For example, if the P4 client form is filled in as follows on client spice, any files from the local depot `depot` will be mapped to `/usr/jake/src/local`, and any files from the remote depot `foo` will be mapped to `/usr/jake/src/remote`.

```
Client: spice
Description:
Created by Jake.
Root:   /usr/jake/src
View:
    //depot/... //spice/local/...
    //foo/... //spice/remote/...
```

- On the left-hand side of any branch or label view, in the same way the mapping has been provided in the above client view.

There is one, rather large, exception to these rules: although files from other local depots can be used in any operation, files from remote depots can be used only in read-only operations. Thus, no files can ever be used in a `p4 submit` to a remote depot.

Additionally, files from remote depots can't be used in `p4 filelog` or `p4 describe`, even though these operations are read-only.

Integrating Files From Other Depots

A branch view may contain remote files in its mappings, so that files can be branched and later integrated from a remote depot into the local one. This works much as local branching and integration do, with two exceptions:

1. When remote files are integrated for the first time (i.e. they don't exist locally), they are opened for `import` rather than `branch`. The difference between `import` and `branch` is only that, upon submission, the remote files are copied locally. Normally, `branch` performs a "lazy copy", referring to the source file/revision until a new revision is submitted. `Import` copies the contents of the source file/revision to the local target.
2. Since it is not possible to make remote files the targets of integration, integrations are one-way only, from remote to local.

Deleting Depots

Depots may be deleted with `p4 -d` *depotname*.

Depot Reporting Commands

All depots known to the current P4D server can be listed with the `p4 depots` command.

APPENDIX A

Environment Variables

Each operating system and shell uses its own syntax for setting environment variables. This table shows how each OS and shell would set the P4CLIENT environment variable.

OS & Shell	Environment Variable Example
UNIX: ksh, sh, bash	`export P4CLIENT=value`
UNIX: csh	`setenv P4CLIENT value`
VMS	`def/j P4CLIENT "value"`
Mac MPW	`set -e P4CLIENT value`

Windows 95/NT	Environment variables can be set with
	`set P4CLIENT=value`,
	but if registry variables are set, they will override the environment variable values. The registry variables can be set in the user-specific part of the registry through the P4 client program with
	`p4 set P4VAR=value`
	and can be set in the system registry with
	`p4 set -s P4VAR=value`
	The use of registry variables is recommended.

The two tables on the next four pages describe the environment variables used by PERFORCE. The most important of the variables are described in the first table; the second table describes the more esoteric variables.

TABLE 1. Basic PERFORCE Environment Variables (page 1 of 2)

Environment Variable	`P4CLIENT`	`P4JOURNAL`	`P4LOG`
Description	Name of client workspace	Database journal file	File to write errors to
Command-Line Alternative	`-c`	`-J`	`-L`
Used by P4?	Yes	No	No
Used by P4d?	No	Yes	Yes
Examples	`eds_elm` `manual`	`journal` `/disk2/perf/journal`	`log` `/disk2/perf/log`
Value if not Explicitly Set	UNIX: name of P4 host NT: value of COMPUTERNAME environment variable	`P4ROOT/journal`	standard error
Notes		File is specified relative to P4D server root, or as an absolute path. Setting this variable, or using the command-line alternative, enables journaling.	File is specified relative to P4D server root, or as an absolute path.

TABLE 2. **Basic PERFORCE Environment Variables** (page 2 of 2)

Environment Variable	`P4PORT`	`P4ROOT`	`P4USER`
Description	For P4D server, port # to listen on. For P4 client, P4D host and its port	Directory in which P4D stores its files and subdirectories	PERFORCE client username
Command-Line Alternative	`-p`	`-r`	`-u`
Used by P4?	Yes	No	Yes
Used by P4D?	Yes	Yes	No

Examples	P4D server example:
	`1515`
	P4 client example:
	`squid:1666`
	`it.com:1308`
		`/usr/lcl/p4root`	`edk`
			`lisag`

Value if not Explicitly Set	For P4D server:
	`1666`
	For P4 client:
	`perforce:1666`
		<none>	UNIX:
			the value of the `USER` environment variable
			NT:
			the value of the `USERNAME` environment variable

Notes	Format on P4 client is `host:port#`, or `port#` by itself if the P4 client and the P4D server run on the same host. To use the default value with P4D, define `perforce` as an alias to the host in `/etc/hosts`, or use the domain name services. Port numbers must be in the range 1024 - 31767.	Create this directory before starting P4D. Only the account running P4D needs read/write permissions in this directory.	By default, the PERFORCE username is the same as the OS username, but this can be set to any other PERFORCE user.

TABLE 3. Esoteric PERFORCE Environment Variables (page 1 of 2)

Environment Variable	`P4DIFF`	`P4EDITOR`	`P4MERGE`
Description	Name and location of the 'diff' program used by `p4 resolve` and `p4 diff`.	Editor used by P4 commands, like `p4 client`, that bring up forms	MERGE program used by `p4 resolve`'s merge command
Command-Line Alternative			
Used by P4?	Yes	Yes	Yes
Used by P4D?	No	No	No

Examples	`diff`
	`diff -b`
	`windiff`
		`vi`
		`emacs`
		`SimpleText`
			Prescient Software's `MergeRight`

Value if not Explicitly Set	UNIX: If DIFF is set, then value of DIFF; otherwise P4's internal *diff* NT: If DIFF is set, then value of DIFF; else if `SHELL` set, `diff`; otherwise, `p4diff.exe`	If EDITOR environment variable is set, its value; otherwise: UNIX: `vi` NT: if `SHELL` is set, `vi`; otherwise, `notepad` VMS: if `POSIX$SHELL` is set, `vi`; otherwise, `edit` Mac: if `EDITOR_SIGNATURE` is set, the program with that four-character creator; otherwise, `SimpleText`.	If `MERGE` environment variable is set, its value; otherwise, nothing.
Notes	This EV can contain flags to diff, such as `diff -u`. `p4 describe`, `p4 diff2`, and `p4 submit` use the *diff* built into P4D; this cannot be changed.		This program is used only by `p4 resolve`'s merge command. It takes four arguments, representing *base*, *theirs*, *yours*, and the resulting *merge* file. Please see page 43 for more details.

TABLE 4. Esoteric PERFORCE Environment Variables (page 2 of 2)

Environment Variable	`P4PAGER`	`PWD`	`TMP, TEMP`
Description	Program used to page output from `p4 resolve`'s `diff.`	The working directory when P4 commands are run.	Directory in which temporary files are written
Command-Line Alternative		`-d`	
Used by P4?	Yes	Yes	Yes
Used by P4D?	No	No	No
Examples	`more`	`/u1/doug/prog`	`/tmp`
Value if not Explicitly Set	If PAGER environment variable is set, then the value of PAGER; otherwise, none.	UNIX: Value of `PWD` as set by shell; if not set by shell, `getwd`() is used. VMS, NT, Mac/MPW: Actual current working directory	UNIX: `/tmp` VMS, Mac/MPW, NT: on client: current directory; on server: P4ROOT
Notes	Used only by `p4 resolve`. If this variable is not set, the output of `p4 resolve`'s *diff* will not be paged.		If the `TEMP` environment variable is set, this is used; otherwise, if `TMP` is set, this is used; otherwise, defaults to the values above.

APPENDIX B

Glossary

access level	A permission given to a user controlling which PERFORCE commands can be used. Access levels are assigned with `p4 protect`. The five access levels are `list`, `read`, `write`, `review`, and `super`.
add	An operation that takes a file in the client workspace and adds it as a new file to the depot.
atomic	Grouping a number of operations together such that either all of them occur, or none of them do. See *atomic change transaction*.
atomic change transaction	Grouping a number of files and operations together in a single changelist, such that when the changelist is submitted, either all the files are updated in the depot, or none of them are.
base	The file revision that two newer, conflicting file revisions were commonly based on.
binary file	A non-ascii file. Whether a file is binary or ascii is determined by the C library call `isascii()`. Binary files are always stored in the depot in full.
branch	Creating a copy of a file or files in the depot so that the new file set can evolve separately from the original files. See *Inter-File Branching*.
branch form	The form displayed when the `p4 branch` command is given.
branch view	The view that specifies how files are mapped from the original codeline to the branched codeline. The mappings within a branch map files within the depot to the copied files within the depot. Branch views are edited in the branch form.
build management	A tool that manages the process of turning source code into product. PERFORCE does not have a build tool, built in, but PERFORCE Software offers a companion freeware build tool called "Jam" at <`www.perforce.com`>.
change	**1.** An edit of a file. **2.** In previous versions of PERFORCE, this term was used as a synonym for *changelist.*
changelist	A list of files, revision numbers of those files, and operations to be performed on these files. The commands `p4 add`, `p4 edit`, `p4 delete`, and `p4 integrate` all include files in a particular changelist, which is sent to the depot atomically when `p4 submit` is typed with no parameters.
changelist form	The form brought up by `p4 change`, and by `p4 submit` when submitting the default change.

changelist number	The number that a particular changelist is known by. The default changelist is assigned a number if a submit of the default changelist fails. Changelist numbers always increase in sequence.
change review	The process of sending email to users when files that they are interested in have changed within the depot.
checkpoint	A copy of the underlying server metadata at one moment in time. This is one half of the journaling process.
client	A computer running P4; a computer storing a client workspace. All PERFORCE work is done by users on client machines. A single PERFORCE system can have many clients, which all talk to a single server via TCP/IP.
client form	The form brought up by the `p4 client` command to define a client workspace.
client name	A name that uniquely identifies the current client workspace. It is set through the `P4CLIENT` environment variable, or on the command-line with the `-c` flag.
client root	The root directory of a client workspace; the lowest level directory under which the managed files sit. The client root is set in the `Root:` field of the client form. If the client name is `proj1`, and the client root is `/usr/joe/project1`, then the file `//proj1/docs/read.me` is actually located on the client machine under `/usr/joe/project1/docs/read.me.`
client side	The right-hand side of a mapping within a client view. Expresses where the corresponding depot files are found within a client workspace.
client view	A set of mappings that express which files from the depot can be accessed from a particular client workspace, and where in the client workspace the depot files are mapped to. Client views are defined in the `View:` field of the form brought up by `p4 client`.
client workspace	A local copy of some or all of the files stored in the depot. These files are managed by PERFORCE; users may work on PERFORCE files only within a client workspace. Client workspaces are defined with the `p4 client` command.
codeline	A set of files that evolve collectively. One codeline can be branched from another, allowing both sets of files to evolve separately from the other.
command line	The interface that this PERFORCE manual describes. If you're looking at this glossary entry because you don't know what a command line is, and you're expecting to use a GUI, you're in the wrong manual.
conflict	See *file conflict*.
counter	A variable tracked and set by `p4 review`. Used internally by PERFORCE to track which changelists have and haven't been reviewed; users can create their own counters for use in their own daemons.
daemon	A program running in the background on the PERFORCE server. PERFORCE provides a change review daemon; users can create their own to handle other needs.
database	Files in a PERFORCE server used to store the server metadata.
default changelist	The changelist used by `p4 add`, `p4 edit`, `p4 delete`, `p4 submit`, etc., unless a numbered changelist is provided to these commands with the `-c` flag. There is always a single default changelist in use for each client workspace.
default depot	The depot that is always available on a PERFORCE server. Its name is `depot`.

delete	1. To remove a file from the client workspace and from the depot with `p4 delete` followed by `p4 submit`. Deleted files are not actually erased from the depot; the head revision of the file is marked as being deleted, but older revisions of the file are still available. 2. To remove an existing client, label, or branch from the PERFORCE server. This is accomplished with the `-d` flag to `p4 client`, `p4 label`, and `p4 branch`.
delta	The line-by-line differences between one file revision and its next (or previous) revision.
delta storage	See *reverse delta storage*.
depot	A file repository on the P4D 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 particular depot. For the default depot `depot`, the depot root is the `depot` subdirectory of the server root directory.
depot side	The left side of any client view mapping. The union of all depot sides of a client view specifies which files are available to the corresponding client workspace.
depot syntax	PERFORCE syntax, when applied to a file in the depot. A file called `readme` in the depot's `doc` subdirectory would be referred to in depot syntax as `//depot/doc/readme`.
detached	A client workspace not connected to a PERFORCE server with a functioning network connection is said to be detached. Permissions must be set on files using OS commands; PERFORCE is unable to open files for you.
difference marker	A named counter used by `p4 review`. Each difference marker keeps track of which changes have already been reviewed for the counter with that name.
distributed depot	A depot within another P4D server, accessed by the local P4D server acting as a proxy client.
donor file	The file from which changes are taken when propagating changes from one codeline to another. A donor file can come either from the branched codeline or the original codeline; this is determined by the use of the `-r` flag to `p4 integrate`.
edit	Opening a file in a client workspace with `p4 edit`. PERFORCE notes that the file has been opened, adds the file revision to a changelist, and turns on write privileges for this file within the client workspace. A file is opened for edit before it is edited with the system editor.
environment variable	A variable set in the operating system's command shell. PERFORCE utilizes environment variables that are passed to the `p4 client` or P4D server.

exclusionary mapping	A view mapping that excludes depot files from a client workspace. Exclusionary mappings begin with a minus sign. For example, to allow a client workspace to access all the files in the depot except for the file called `secret`, the view might look like this:
	`//depot/...              //a_client/...`
	` -//depot/secret         //a_client/secret`

exclusionary access	A permission that denies access on the specified files. For example, the following permission denies Ed `write` permissions on all files in the directory `secret`: `write     edk     *     -//depot/secret`
fast	See PERFORCE.

file	In a client workspace, `file` has the usual meaning. A file in the depot consists of the head revision of the file, and every revision of the same file.
file conflict	A state in which the version of a file in the client workspace is not an edit of the head revision in the depot at submit time.
file pattern	An argument on a P4 command line specifying files using wildcards.
file reference	A file revision specification, like foo#3. Used specifically when storing pointers to files in lists such as a label. The label contains file references, not the contents of the files themselves.
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, like `foo#3`.
file tree	All the subdirectories and files under a given root directory.
file type	An attribute that determines how a particular file is handled by PERFORCE. The two basic PERFORCE file types are `text` and `binary`, but there are quite a few subtypes.
fix	A job that has been linked to a changelist with `p4 fix`, `p4 submit`, or `p4 change`. Under most circumstances, a fixed job has a status of `closed`, but this is not always true: when an open job is linked to a pending changelist, it is still open, and closed, fixed jobs can always be reopened or suspended.
form	Screens brought up by certain PERFORCE commands containing elements whose value needs to be changed. The form is displayed in an external editor; the editor used is defined by the environment variable `P4EDIT`. The commands that bring up forms are `p4 change`, `p4 client`, `p4 depot`, `p4 job`, `p4 label`, `p4 user`, and sometimes `p4 submit.`
full-file storage	The method by which PERFORCE stores file revisions of binary files within the depot: every file revision is stored in full. Contrast this with *reverse delta storage*, which is used for text files.
get	Formerly used to describe the process accomplished by `p4 get`, which has been renamed `p4 sync`. The `p4 get` command can still be used as a synonym for `p4 sync`. See `sync`.
GNU	Software provided by the Free Software Foundation. Most GNU software is system level software such as enhanced versions of standard UNIX utilities and language compilers. GNU stands for "GNU's not UNIX".
have list	The list of file revisions that the PERFORCE server believes are currently in the client workspace. Generated by `p4 have`.
head revision	The most recent revision of a file within the depot. Since file revisions are numbered sequentially, this will also be the highest-numbered revision of the file. To refer to the head revision of file `foo`, use `foo#head`.
integrate	To propagate changes from one codeline to another.
integration record	The data structure by which PERFORCE keeps track of integrated files. It tracks which revisions of which donor files were integrated into which target files.

Inter-File Branching	PERFORCE's branching mechanism. It differs from the branching mechanism used by most SCM systems, allowing arbitrary copies of files to be stored anywhere in the depot, and allowing these files to evolve separately. Additionally, changes made to any file can be propagated to the corresponding file in any branch.
job	A generic term for a defect report, system improvement request, or change order. An arbitrary textual description of some change intended to be made to the system.
job tracking	PERFORCE's mechanism for keeping track of jobs. It is implemented via `p4 job` and `p4 fix`.
journal	A file containing a record of every change made to the PERFORCE server's metadata since the time of the last checkpoint. One half of the journaling process.
journaling	Keeping track of every change made to the PERFORCE server's metadata since one particular moment in time. Requires a checkpoint file and a journal file. Only the server's metadata is journaled; external processes need to be run to backup the depot's file revisions.
label	A user-configurable list of file revisions. Used to save "important" file configurations for later use. Any file included in a label can be referred to with the revision specifier `@labelname`; e.g. `foo@release3.0.1.`
label view	The view that defines which files in the depot are stored in a particular label. This can be edited in the form brought up by `p4 label`. The left-hand side of each mapping represents a subtree of the depot's file tree; the right side represents the files' corresponding location within the label.
license	PERFORCE's mechanism to ensure that our software is run on each site only by the number of users that have been paid for. Two users may be run on any PERFORCE server without a license or any form of payment. For more information on licensing, please send email to `info@perforce.com`.
list access	A protections level giving the user permission to run reporting commands, such as `p4 client`, that give access to metadata. A user with only list access to a particular file can't run any commands that would allow them to read or write the contents of the file.
local depot	Any depot located on the current PERFORCE server. By default, only one such depot is constructed; others may be defined with `p4 depot`.
local syntax	The native name of a file on the client host, as would be used by other commands in that operating system. For example, file `foo` in joe's home directory in UNIX local syntax would be `~joe/foo`, or `/usr/joe/foo`, etc.
lock	Locking a file with `p4 lock` ensures that no other clients will be able to submit the same file until the file is unlocked by the locking client. Files are automatically locked when submission starts, and unlocked when submission ends; they can be manually unlocked with `p4 unlock`.
log	Error output from the P4D server. By default, this is written to standard error; a specific file can be set in the `P4LOG` environment variable, or with the `-L` flag to P4D.
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. (See also *client view, branch view, label view*).
merge	To combine the contents of two conflicting file revisions into a single file. This is accomplished with `p4 resolve`.

merge file	A file revision generated by PERFORCE from two conflicting file revisions. This file can be edited by the user during the `p4 resolve` process, producing a result acceptable to the user.
metadata	The data stored by the server that describes the files in the depot, the current state of client workspaces, protections, users, clients, labels, and branches. It includes all the data stored in the server except for the actual contents of the files.
modtime	The modification time of a file that has been read from the depot into a client workspace. By default, the modtime is the time that the file was last written to the depot; this can be changed in the P4 client form to be the time that the file is read into the client workspace.
namespace	The pool of legal names for clients, branches, depots, and labels. Clients, branches, depots, and labels all share the same namespace; therefore a client cannot have the same name as any branch, depot, or label.
network connection	A TCP/IP connection between a PERFORCE client and PERFORCE server. Generally, users are expected to work within a functioning network connection; they can still edit client files without a network connection by following the instructions for working detached.
nonexistent revision	A special file revision; a completely empty revision of any file. Useful only to remove a file from the client workspace while leaving it intact in the depot: use `p4 sync foo#none`.
numbered changelist	A changelist that has been created by PERFORCE but that has not yet been submitted. A numbered changelist can be created manually, with `p4 change -c`, or is created automatically by PERFORCE when a submit of the default change has failed. Once a changelist has been assigned a number, it must be referred to by that number in all subsequent commands, e.g. `p4 submit -c 31`.
open	A file in a client workspace that has been included on a changelist with `p4 add`, `p4 delete`, or `p4 edit`. The file is said to be open within the client workspace.
owner	The PERFORCE user who created a particular client, branch, or label. This can be changed through the form brought up by `p4 client`, `p4 branch`, or `p4 label`.
P4	The program invoked by users from a client to run all PERFORCE commands. It talks via TCP/IP to the PERFORCE server, mediating the interaction between the managed files in the client workspace and the master repository and metadata on the server host.
P4D	The program on the PERFORCE server that manages the depot and the metadata. It waits on a TCP/IP port for a connection from the client program.
pending change	An existing changelist that has not yet been submitted. These can be created manually with `p4 change -c`, and are automatically created when submission of the default change fails.
PERFORCE	The *fast* Software Configuration Management system.
PERFORCE server	See *server*.
PERFORCE syntax	A syntax for referring to files that remains invariant across operating systems. It consists of two slashes, followed by the depot or client name, followed by a slash, and then the name of the file specified relative to the depot or client root.
Perl	A scripting language available for most operating systems. Useful for creating programs that utilize PERFORCE commands, such as the change review daemon.
permission	See *access*.

port	A TCP/IP logical channel. Since every application on a host listens to a different port, the ports are used to funnel messages to the correct application. The P4D server program listens on the port assigned by P4PORT.
project	Generic term for some set of files kept by PERFORCE. One server might be storing source files for multiple projects.
propagate	To copy changes in one file to a branched copy of the same file, leaving the rest of the branch file intact.
protections	The complete set of permissions as stored in the server's protections table.
pure integration	An integration in which changes to the target file incorporates only revisions from a single source file.
RCS format	Revision Control System algorithm and data structure for storing file revisions. Uses reverse delta encoding for file storage. This is the method used by PERFORCE. See also *reverse delta encoding.*
read access	A protections level giving the user permission to run commands, such as `p4 sync`, that allow them to read the contents of PERFORCE files stored in the depot. Read access includes list access.
refresh	To copy the contents of an unopened file from the depot into the client workspace.
remote depot	A depot located on a server other than the current PERFORCE server defined under P4PORT. PERFORCE superusers can make these depots accessible to the current server's clients for read-only access.
renumber	PERFORCE may renumber a changelist when it is submitted. Since change numbers are allocated sequentially, a change might have one number when it is created, and another when it is submitted.
reresolve	To run the resolve process a second time. This can be done only between the time a file is resolved and the time it is submitted
resolve	The process by which an integration is finalized by the user. The resolve process, run by `p4 resolve`, allows the user to decide whether to keep the integrated file, edit it, or accept some other revision of the file in place of the integrated revision.
resource fork	One fork of a Macintosh file. The PERFORCE file type is `resource`.
reverse delta storage	The method by which PERFORCE stores file revisions of text files within the depot. Rather than store every file revision in full, PERFORCE stores only the deltas from each revision to the one previous, storing the full text of only the head revision.
revert	To throw away a file in the client workspace, replacing it with the revision in the depot that was being edited. Files that were opened with `p4 add` are left in the client workspace; they are simply removed from the corresponding changelist. Reverting files can only be done before the files have been submitted.
review access	A special protections level given to the review daemon, or to any daemon created by a user. It includes read and list accesses, plus permission to run the `p4 review` command.
review daemon	Any daemon process written that uses the `p4 review` command. See also *change review.*

review marker	Any named counter used by `p4 review`. Counter values are stored as PERFORCE metadata, so their value remains the same from execution to execution of `p4 review`. Each review marker has its own name, hence its own value. The value of any counter can be accessed and set at any time. Review markers are commonly used to track which changelists have been processed by a particular daemon, but other uses are possible.
revision	A specific version of a file within the depot.
revision number	A number indicating which revision of the file is being referred to. Revision numbers start at 1, and increase sequentially. To refer to a particular revision of a file, append the pound sign and the revision number to the name of the file; e.g., the tenth revision of file `foo` would be referred to as `foo#10.` The revision specification foo#10 is actually the same as the revision range specification foo#1,#10, since revision ten of the file encompasses all the changes made in the file from revision 1 to revision 10.
revision range	A range of revision numbers for a specified file, specified as the low and high end of the range. For example, `foo#5,7` would access the fifth through seventh revisions of file `foo`. Only a few commands allow revision ranges to be specified. The only non-reporting command that allows a revision range is `p4 integrate`.
revision specification	A suffix to filenames that specify a particular revision of that file. Revision specifiers can refer to files by revision number, change number, label name or client name.
root	A top level directory under which all accessible files are found. See *client root*, *server root*, *depot root*.
SCM	Commonly used abbreviation for Software Configuration Management.
Sendmail	A UNIX mail transfer agent that, like a post office, collects mail and figures out how to move it further along. Used by the PERFORCE change review daemon.
server	The PERFORCE depot and metadata on a central UNIX or NT host. All the client workspaces in a PERFORCE system access the same server.
server root	The directory in which the server program stores its metadata and all the shared files. The directory is set via the P4ROOT environment variable.
Software Configuration Management	A category of software whose definition changes from manufacturer to manufacturer. PERFORCE is a Software Configuration Management system. Functions commonly said to comprise SCM systems are version control, concurrent development, release management, build management, and change review.
status	For a changelist, a value that indicates whether the changelist is `new`, `pending` or `submitted`. For a job, an indicator of whether the job is `open`, `closed`, or `suspended`.
submit	To send a pending changelist to the server for processing. The files referenced by the changelist are locked, the corresponding operations are performed, and then the files are unlocked.
subscribe	To register to receive email whenever changelists are submitted that affect particular files. Files are subscribed to via the `p4 user` form; the change review daemon watches the depot and sends email when the specified files have been affected.
superuser	A PERFORCE user with superuser permissions.

superuser access	A protections level that gives the user permission to run every PERFORCE command, including those that set protections, obliterate files, verify files with an MD5 signature, and set up connections to remote depots.
sync	To copy a file revision (or set of file revisions) from the depot to a client workspace. This is accomplished with the `p4 sync` command.
target file	When integrating changes between a branched codeline and the original codeline, the target file is the file that receives the changes from the donor file. Either the branched file or the original file can be the target; this is determined by the use or non-use of the `-r` flag when calling `p4 integrate`.
TCP/IP	A networking protocol; the protocol of the Internet.
text file	The basic PERFORCE file type. Text files can be stored using reverse delta encoding, saving space on the server.
theirs	When resolving a file conflict, the *theirs* file is the revision in the depot that the client file conflicts with. This is usually the head revision. When working with branched files, theirs is the *donor* file.
three-way merge	A merge between *yours*, *theirs*, and *base*. When a three-way merge is schedule by PERFORCE, it is because *yours* and *theirs* are both revisions of a common *base* file, theirs has been submitted to the depot, and the unconditional acceptance of *yours* would lose all the changes to *theirs*.
tip revision	A term sometimes used by other SCM systems in place of *head revision*.
two-way merge	The type of merge performed when a donor file is being integrated into a target file that was branched from a different file. In this case, there is no *base* file, so a two-way merge is performed, in which your file is used as the *base* of the merge. In a two-way merge, all changes appear as *theirs*, and there can be no conflicts.
user	An identifier that informs PERFORCE who is running the P4 commands. By default, this is the same as the system username, but the environment variable `P4USER` overrides this, allowing any user to impersonate any other.
version control	The tracking of each revision and variant of every file managed by an SCM system
view	A description of the relationship between files in the depot and a client workspace, label, or branch. The view determines which files from the depot are included in the mapping, and the names by which the mapped files are known. See *client view*, *label view*, *branch view*.
wildcard	A special character that is not interpreted literally, but is used to match other characters in strings. PERFORCE wildcards are '`*`', which matches anything except a slash; '`...`', which matches any string including slashes, and '`%d`', which is used for parametric substitution in views.
workspace	See *client workspace*.
write access	A protections level that gives the user permission to run commands, such as `p4 submit` and `p4 edit`, that allow him to alter the contents of files in the depot. Write access includes `read` and `list` accesses.
yours	When resolving a file conflict, the *yours* file is the edited version within the client workspace. In an integration of a branched file, *yours* is the *target* file.

Index