Perforce in FreeBSD Development

Scott Long

scottl@FreeBSD.org

$FreeBSD: release/8.4.0/en_US.ISO8859-1/articles/p4-primer/article.xml 41130 2013-03-07 23:37:00Z gavin $
$FreeBSD: release/8.4.0/en_US.ISO8859-1/articles/p4-primer/article.xml 41130 2013-03-07 23:37:00Z gavin $

Table of Contents

1 Introduction

The FreeBSD project uses the **Perforce** version control system to manage experimental projects that are not ready for the main Subversion repository.

1.1 Availability, Documentation, and Resources

While **Perforce** is a commercial product, the client software for interacting with the server is freely available from Perforce. It can be easily installed on FreeBSD via the `devel/p4` port or can be downloaded from the **Perforce** web site at http://www.perforce.com/perforce/loadprog.html, which also offers client applications for other OS's.

While there is a GUI client available, most people use the command line application called `p4`. This document is written from the point of view of using this command.

Detailed documentation is available online at http://www.perforce.com/perforce/technical.html.

Reading the "Perforce User's Guide" and "Perforce Command Reference" is highly recommended. The **p4** application also contains an extensive amount of online help accessible via the `p4 help` command.

The FreeBSD **Perforce** server is hosted on `perforce.freebsd.org`, port `1666`. The repository is browsable online at http://p4web.freebsd.org. Some portions of the repository are also automatically exported to a number of legacy **CVSup** servers.

2 Getting Started

The first step to using **Perforce** is to obtain an account on the server. If you already have a `FreeBSD.org` account, log into `freefall`, run the following command, and enter a password that is not the same as your FreeBSD login or any other SSH passphrase:

```
% /usr/local/bin/p4newuser
```

Of course if you do not have a `FreeBSD.org` account, you will need to coordinate with your sponsor.

> **Warning:** An email will be sent to your FreeBSD address that contains the password you specified above in cleartext. Be sure to change the password once your **Perforce** account has been created!

The next step is to set the environment variables that `p4` needs, and verify that it can connect to the server. The `P4PORT` variable is required to be set for all operations, and specifies the appropriate **Perforce** server to talk to. For the FreeBSD project, set it like so:

```
% export P4PORT=perforce.freebsd.org:1666
```

> **Note:** Users with shell access on the `FreeBSD.org` cluster may wish to tunnel the **Perforce** client-server protocol via an SSH tunnel, in which case the above string should be set to `localhost`.

The FreeBSD server also requires that the `P4USER` and `P4PASSWD` variables be set. Use the username and password from above, like so:

```
% export P4USER=username
% export P4PASSWD=password
```

Test that this works by running the following command:

```
% p4 info
```

This should return a list of information about the server. If it does not, check that you have the `P4PORT` variable set correctly.

3 Clients

Perforce provides access to the repository and tracks state on a per-client basis. In **Perforce** terms, a client is a specification that maps files and directories from the repository to the local machine. Each user can have multiple clients, and each client can access different or overlapping parts of the repository. The client also specifies the root directory of the file tree that it maps, and it specifies the machine that the tree lives on. Thus, working on multiple machines requires that multiple clients be used.

Clients may be accessed via the `p4 client` command. Running this command with no arguments will bring up a client template in an editor, allowing you to create a new client for your work. The important fields in this template are explained below:

`Client:`

This is the name of the client spec. It can be anything you want, but it must be unique within the **Perforce** server. A naming convention that is commonly used is *`username_machinename`*, which makes it easy to identify clients when browsing them. A default name will be filled in that is just the machine name.

`Description:`

This can contain a simple text description to help identify the client.

`Root:`

This is the local directory that will serve as the root directory of all the files in the client mapping. This should be a unique location in your filesystem that does not overlap with other files or **Perforce** clients.

`Options:`

Most of the default options are fine, though it is usually a good idea to make sure that the `compress` and `rmdir` options are present and do not have a `no` prefix on them. Details about each option are in the **Perforce** docs.

`LineEnd:`

This handles CR-LF conversions and should be left to the default unless you have special needs for it.

`View:`

This is where the server-to-local file mappings go. The default is

`//depot/... //`*`client`*`/...`

This will map the entire **Perforce** repository to the `Root` directory of your client. *DO NOT USE THIS DEFAULT!* The FreeBSD repo is huge, and trying to map and sync it all will take an enormous amount of resources. Instead, only map the section of the repo that you intend to work on. For example, there is the smpng project tree at `//depot/projects/smpng`. A mapping for this might look like:

`//depot/projects/smpng/... //`*`client`*`/...`

The `...` should be taken literally. It is a **Perforce** idiom for saying "this directory and all files and directories below it."

A Perforce "view" can contain multiple mappings. Let's say you want to map in both the SMPng tree and the NFS tree. Your View might look like:

```
//depot/projects/smpng/... //client/smpng/...
        //depot/projects/nfs/... //client/nfs/...
```

Remember that the *client* is the name of the client that was specified in the `Client` section, but in the `View` it also resolves to the directory that was specified in the `Root` section.

Also note that the same file or directory cannot be mapped multiple times in a single view. The following is illegal and will produce undefined results:

```
//depot/projects/smpng/... //client/smpng-foo/...
        //depot/projects/smpng/... //client/smpng-bar/...
```

Views are a tricky part of the learning experience with **Perforce**, so do not be afraid to ask questions.

Existing clients can be listed via the `p4 clients` command. They can be viewed without being modified via the `p4 client -o` *clientname* command.

Whenever you are interacting with files in **Perforce**, the `P4CLIENT` environment variable must be set to the name of the client that you are using, like so:

```
% export P4CLIENT=myclientname
```

Note that client mappings in the repository are not exclusive; multiple clients can map in the same part of the repository. This allows multiple people to access and modify the same parts of the repository, allowing a team of people to work together on the same code.

4 Syncing

Once you have a client specification defined and the `P4CLIENT` variable set, the next step is to pull the files for that client down to your local machine. This is done with the `p4 sync` command, which instructs **Perforce** to synchronize the local files in your client with the repository. The first time it runs, it will download all of the files. Subsequent runs will only download files that have changed since the previous run. This allows you to stay in sync with others whom you might be working with.

Sync operations only work on files that the **Perforce** server knows has changed. If you change or delete a file locally without informing the server, doing a sync will not bring it back. However, doing a `p4 sync -f` will unconditionally sync all files, regardless of their state. This is useful for resolving problems where you think that your tree might be corrupt.

You can sync a subset of your tree or client by specifying a relative path to the sync command. For example, to only sync the `ufs` directory of the `smpng` project, you might do the following:

```
% cd projectroot/smpng
% p4 sync src/sys/ufs/...
```

Specifying a local relative path works for many other `p4` commands.

5 Branches

One of the strongest features of **Perforce** is branching. Branches are very cheap to create, and moving changes between related branches is very easy (as will be explained later). Branches also allow you to do very experimental work in a sandbox-like environment, without having to worry about colliding with others or destabilizing the main tree. They also provide insulation against mistakes while learning the **Perforce** system. With all of these benefits, it makes sense for each project to have its own branch, and we strongly encourage that with FreeBSD. Frequent submits of changes to the server are also encouraged.

sSimilar to **Subversion**, the **Perforce** repository (the "depot") is a single flat tree. Every file, whether a unique creation or a derivative from a branch, is accessible via a simple path under the server `//depot` directory. When you create a branch, all you are doing is creating a new path under the `//depot`. This is in sharp contrast to systems like CVS, where each branch lives in the same path as its parent. With **Perforce**, the server tracks the relationship between the files in the parent and child, but the files themselves live under their own paths.

The first step to creating a branch is to create a branch specification. This is similar to a client specification, but is created via the command `p4 branch` *`branchname`*.

The following important fields are explained:

`Branch`

: The name of the branch. It can be any name, but must be unique within the repository. The common convention in FreeBSD is to use *`username_projectname`*.

`Description`

: This can hold a simple text description to describe the branch.

`View`

: This is the branch mapping. Instead of mapping from the depot to the local machine like a client map, it maps between the branch parent and branch child in the depot. For example, you might want to create a branch of the smpng project. The mapping might look like:

  ```
  //depot/projects/smpng/... //depot/projects/my-super-smpng/...
  ```

  Or, you might want to create a brand new branch off of the stock FreeBSD sources:

  ```
  //depot/vendor/freebsd/... //depot/projects/my-new-project/...
  ```

  This will map the FreeBSD HEAD tree to your new branch.

Creating the branch spec only saves the spec itself in the server, it does not modify the depot or change any files. The directory that you specified in the branch is empty on the server until you populate it.

To populate your branch, first edit your client with the `p4 client` command and make sure that the branch directory is mapped in your client. You might need to add a `View` line like:

```
//depot/projects/my-new-project/... //myclient/my-new-project/...
```

The next step is to run the `p4 integrate` command, as described in the next section.

6 Integrations

"Integration" is the term used by **Perforce** to describe the action of moving changes from one part of the depot to another. It is most commonly done in conjunction with creating and maintaining branches. An integration is done when you want to initially populate a branch, and it is done when you want to move subsequent changes in the branch from the parent to the child, or from the child to the parent. A common example of this is periodically integrating changes from the vendor FreeBSD tree to your child branch tree, allowing you to keep up to date with changes in the FreeBSD tree. The **Perforce** server tracks the changes in each tree and knows when there are changes that can be integrated from one tree to another.

The common way to do an integration is with the following command:

```
% p4 integrate -b branchname
```

branchname is the name given to a branch spec, as discussed in the previous section. This command will instruct **Perforce** to look for changes in the branch parent that are not yet in the child. From those changes it will prepare a list of diffs to move. If the integration is being done for the first time on a branch (i.e. doing an initial population operation), then the parent files will simply be copied to the child location on the local machine.

Once the integration operation is done, you must run `p4 resolve` to accept the changes and resolve possible conflicts. Conflicts can arise from overlapping changes that happened in both the parent and child copy of a file. Usually, however, there are no conflicts, and **Perforce** can quickly figure out how to merge the changes together. Use the following commands to do a resolve operation:

```
% p4 resolve -as
% p4 resolve
```

The first invocation will instruct **Perforce** to automatically merge the changes together and accept files that have no conflicts. The second invocation will allow you to inspect each file that has a possible conflict and resolve it by hand if needed.

Once all of the integrated files have been resolved, they need to be committed back to the repository. This is done via the `p4 submit` command, explained in the next section.

7 Submit

Changes that are made locally should be committed back to the **Perforce** server for safe keeping and so that others can access them. This is done via the `p4 submit` command. When you run this command, it will open up a submit template in an editor. FreeBSD has a custom template, and the important fields are described below:

```
Description:
        <enter description here>
        PR:
        Submitted by:
        Reviewed by:
        Approved by:
        Obtained from:
        MFP4 after:
```

It is good practice to provide at least 2-3 sentences that describe what the changes are that you are submitting. You should say what the change does, why it was done that way or what problem is solves, and what APIs it might

change or other side effects it might have. This text should replace the `<enter description here>` line in the template. You should wrap your lines and start each line with a TAB. The tags below it are FreeBSD-specific and can be removed if not needed.

`Files:`

This is automatically populated with all of the files in your client that were marked in the add, delete, integrate, or edit states on the server. It is always a very good idea to review this list and remove files that might not be ready yet.

Once you save the editor session, the submit will happen to the server. This also means that the local copies of the submitted files will be copied back to the server. If anything goes wrong during this process, the submit will be aborted, and you will be notified that the submit has been turned into a changelist that must be corrected and re-submitted. Submits are atomic, so if one file fails, the entire submit is aborted.

Submits cannot be reverted, but they can be aborted while in the editor by exiting the editor without changing the `Description` text. **Perforce** will complain about this the first time you do it and will put you back in the editor. Exiting the editor the second time will abort the operation. Reverting a submitted change is very difficult and is best handled on a case-by-case basis.

8 Editing

The state of each file in the client is tracked and saved on the server. In order to avoid collisions from multiple people working on the same file at once, **Perforce** tracks which files are opened for edit, and uses this to help with submit, sync, and integration operations later on.

To open a file for editing, use the `p4 edit` command like so:

```
% p4 edit filename
```

This marks the file on the server as being in the *edit* state, which then allows it to be submitted after changes are made, or marks it for special handling when doing an integration or sync operation. Note that editing is not exclusive in **Perforce**. Multiple people can have the same file in the edit state (you will be informed of others when you run the `edit` command), and you can submit your changes even when others are still editing the file.

When someone else submits a change to a file that you are editing, you will need to resolve his changes with yours before your submit will succeed. The easiest way to do this is to either run a `p4 sync` or `p4 submit` and let it fail with the conflict, then run `p4 resolve` to manually resolve and accept his changes into your copy, then run `p4 submit` to commit your changes to the repository.

If you have a file open for edit and you want to throw away your changes and revert it to its original state, run the `p4 revert` command like so:

```
% p4 revert filename
```

This resyncs the file to the contents of the server, and removes the edit attribute from the server. Any local changes that you had will be lost. This is quite useful when you have a made changes to a file but later decide that you do not want to keep them.

When a file is synced, it is marked read-only in the filesystem. When you tell the server to open it for editing, it is changed to read-write on the filesystem. While these permissions can easily be overridden by hand, they are meant to gently remind you that you should being using the `p4 edit` command. Files that have local changes but are not in the edit state may get overwritten when doing a `p4 sync`.

9 Changes, Descriptions, and History

Changes to the **Perforce** depot can be listed via the `p4 changes` command. This will provide a brief description of each change, who made the change, and what its change number was. A change can be examined in detail via the `p4 describe` *`changenumber`* command. This will provide the submit log and the diffs of the actual change.

Commonly, the `p4 describe` command is used in one of three ways:

`p4 describe -s` *`CHANGE`*

> List a short description of changeset *CHANGE*, including the commit log of the particular changeset and a list of the files it affected.

`p4 describe -du` *`CHANGE`*

> List a description of changeset *CHANGE*, including the commit log of the particular changeset, a list of the files it affected and a patch for each modified file, in a format similar to "unified diff" patches (but not exactly the same).

`p4 describe -dc` *`CHANGE`*

> List a description of changeset *CHANGE*, including the commit log of the particular changeset, a list of the files it affected and a patch for each modified file, in a format similar to "context diff" patches (but not exactly the same).

The `p4 filelog` *`filename`* command will show the history of a file, including all submits, integrations, and branches of it.

10 Diffs

There are two methods of producing file diffs in **Perforce**, either against local changes that have not been submitted yet, or between two trees (or within a branch) in the depot. These are done with different commands, `diff` and `diff2`:

`p4 diff`

> This generates a diff of the local changes to files in the edit state. The `-du` and `-dc` flags can be used to create unified or context diffs, respectively, or the `P4DIFF` environment variable can be set to a local diff command to be used instead. It is a very good idea to use this command to review your changes before submitting them.

`p4 diff2`

> This creates a diff between arbitrary files in the depot, or between files specified in a branch spec. The diff operation takes place on the server, so `P4DIFF` variable has no effect, though the `-du` and `-dc` flags do work. The two forms of this command are:
>
> `%` **`p4 diff2 -b`** ***`branchname`***
>
> and
>
> `%` **`p4 diff2 //depot/`*`path1`*** **`//depot/`*`path2`***

In all cases the diff will be written to the standard output. Unfortunately, **Perforce** produces a diff format that is slightly incompatible with the traditional Unix diff and patch tools. Using the `P4DIFF` variable to point to the real diff(1) tool can help this, but only for the `p4 diff` command. The output of `diff2` command must be

post-processed to be useful (the `-u` flag of `diff2` will produce unified diffs that are somewhat compatible, but it does not include files that have been added or deleted). There is a post-processing script at: http://people.freebsd.org/~scottl/awkdiff.

11 Adding and Removing Files

Integrating a branch will bring existing files into your tree, but you may still want to add new files or remove existing ones. Adding files is easily done be creating the file and then running the `p4 add` command like so:

```
% p4 add filename
```

If you want to add a whole tree of files, run a command like:

```
% find . -type f | xargs p4 add
```

> **Note: Perforce** can track UNIX symlinks too, so you can probably use "`\! -type d`" as the matching expression in find(1) above. We don't commit symlinks into the source tree of FreeBSD though, so this should not be necessary.

Doing a `p4 submit` will then copy the file to the depot on the server. It is very important to only add files, not directories. Explicitly adding a directory will cause **Perforce** to treat it like a file, which is not what you want.

Removing a file is just as easy with the `p4` delete command like so:

```
% p4 delete filename
```

This will mark the file for deletion from the depot the next time that a submit is run. It will also remove the local copy of the file, so beware.

Of course, deleting a file does not actually remove it from the repository.

Deleted files can be resurrected by syncing them to a prior version. The only way to permanently remove a file is to use the `p4 obliterate` command. This command is irreversible and expensive, so it is only available to those with admin access.

12 Working with diffs

Sometimes you might need to apply a diff from another source to a tree under **Perforce** control. If it is a large diff that affects lots of files, it might be inconvenient to manually run `p4 edit` on each file. There is a trick for making this easier. First, make sure that no files are open on your client and that your tree is synced and up to date. Then apply the diff using the normal tools, and coerce the permissions on the files if needed. Then run the following commands:

```
% p4 diff -se ... | xargs p4 edit
% p4 diff -sd ... | xargs p4 delete
% find . -type f | xargs p4 add
```

The first command tells **Perforce** to look for files that have changed, even if they are not open. The second command tells **Perforce** to look for files that no longer exist on the local machine but do exist on the server. The third command

then attempts to add all of the files that it can find locally. This is a very brute-force method, but it works because **Perforce** will only add the files that it does not already know about. The result of running these commands will be a set of files that are opened for edit, removal, or add, as appropriate.

Verify the active changelist with:

```
% p4 changelist
% p4 diff -du
```

and just do a `p4 submit` after that.

13 Renaming files

Perforce does not have a built-in way of renaming files or moving them to a different part of the tree. Simply copying a file to the new location, doing a `p4 add` on it, and a `p4 delete` on the old copy, works, but does not preserve change history of the file. This can make future integrations with parents and children very bumpy, in fact. A better method of dealing with this is to do a one-time, in-tree integration, like so:

```
% p4 integrate -i oldfile newfile
% p4 resolve
% p4 delete oldfile
% p4 submit
```

The integration will force **Perforce** to keep a record of the relationship between the old and new names, which will assist it in future integrations. The `-i` flag tells it that it is a "baseless" integration, meaning that there is no branch history available for it to use in the integration. That is perfect for an integration like this, but should not be used for normal branch-based integrations.

14 Interactions between FreeBSD Subversion and Perforce

The FreeBSD **Perforce** and **Subversion** repositories are completely separate. However, changes to Subversion are tracked at near-real-time in **Perforce**. Every 2 minutes, the Subversion server is polled for updates in the HEAD branch, and those updates are committed to **Perforce** in the `//depot/vendor/freebsd/...` tree. This tree is then available for branching and integrating to derivative projects. Any project that directly modifies that FreeBSD source code should have this tree as its branch parent (or grandparent, depending on the needs), and periodic integrations and syncs should be done so that your tree stays up to date and avoids conflicts with mainline development.

The bridge between Subversion and **Perforce** is one-way; changes to Subversion will be reflected in **Perforce**, but changes in Perforce will not be reflected in Subversion. On request, some parts of the **Perforce** repo can be exported to **CVSup** and made available for distribution that way. Contact the FreeBSD **Perforce** administrators if this is something that you are interested in.

15 Offline Operation

One weakness of **Perforce** is that it assumes that network access to the server is always available. Most state, history, and metadata is saved on the server, and there is no provision for replicating the server like there is with CVS/**CVSup**. It is possible to run a proxy server, but it only provides very limited utility for offline operation.

The best way to work offline is to make sure that your client has no open files and is fully synced before going offline. Then when editing a file, manually change the permissions to read-write. When you get back online, run the commands listed in the Section 12 to automatically identify files that have been edited, added, and removed. It is quite common to be surprised by **Perforce** overwriting a locally changed file that was not opened for edit, so be extra vigilant with this.

16 Notes for Google Summer of Code

Most FreeBSD projects under the Google Summer of Code program are located on the FreeBSD **Perforce** server under one of the following locations:

- `//depot/projects/soc2005/`*`project-name`*`/...`
- `//depot/projects/soc2006/`*`project-name`*`/...`
- `//depot/projects/soc2007/`*`project-name`*`/...`
- `//depot/projects/soc2008/`*`project-name`*`/...`

The project mentor is responsible for choosing a suitable project name and getting the student going with **Perforce**.

Access to the FreeBSD **Perforce** server does not imply membership in the FreeBSD CVS committer community, though we happily encourage all students to consider joining the project when the time is appropriate.