HOW BASH REDIRECTION WORKS UNDER THE HOOD

Have you ever wondered how bash redirection works under the hood? Redirection itself is pretty straightforward.

Using bash, you can redirect a file to a process' stdin, or redirect a process' stdout/stderr to a file or other file descriptor (including things like redirecting stderr to stdout, because both are file descriptors).

Redirection looks like this:

```
ls -la > output.txt
```

The above command redirects stdout from the ls command to the file output.txt.

Here's how you redirect stderr:

```
ls -la 2> errors.txt
```

This will send everything from stderr to errors.txt (such as permission denied messages). That's why for example, if you run:

```
ls /root > /dev/null
```

As a non-root user, you still see an error message, even though you redirected the output to `/dev/null`. Errors are normally written to stderr.

Finally, you can also do input redirection for files that take stdin as output.

```
bash < commands.txt
```

The above command will read the contents of commands.txt and execute them using the bash interpreter.

Under the hood

Anyway, the point of this post is how such a feature is implemented. It turns out to be exceedingly simple if you understand process forking.

When a process wants to execute another process (e.g. bash running ls), it generally works like this:

1. The main process (e.g. bash) forks itself using the fork glibc wrapper (sidenode: In c, when you call `fork`, you're actually calling the glibc wrapper, not the `fork(2)` syscall. The glibc wrapper is implemented using the `clone(2)` syscall, not the `fork(2)` syscall, as clone is more powerful)
2. The forked process sees that output redirection was entered on the command line and opens the specified file using the `open(2)` syscall or something equivalent
3. The forked process calls `dup2` to copy the freshly opened file descriptor over stdin/stdout/stderr
4. The forked process closes the original file handler to avoid resource leaks
5. The forked process proceeds as normal by calling the `execve(2)` syscall or something similar to replace the executable image with that of the process to be run (e.g. ls)

Here’s a code example that does the same thing:

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>


int main(void)
{
   char *argv[] = { "/bin/ls", "-la", 0 };
   char *envp[] =
   {
      "HOME=/",
      "PATH=/bin:/usr/bin",
      "USER=brandon",
      0
   };
   int fd = open("/home/brandon/ls.log", O_WRONLY|O_CREAT|O_TRUNC, 0666);
   dup2(fd, 1); // stdout is file descriptor 1
   close(fd);
   execve(argv[0], &argv[0], envp);
   fprintf(stderr, "Oops!\n");
   return -1;
}
```

The above code will set stdout to ls.log and than run “ls -la”.

Please note that fprintf and below won’t get executed unless `execve` fails.

And that’s how redirection is implemented in bash!

Source: http://brandonwamboldt.ca/how-bash-redirection-works-under-the-hood-1512/