Process substitution

Process substitution is a form of redirection where the input or output of a process (some sequence of commands) appear as a temporary file.

```
<( <LIST> )

>( <LIST> )
```

Process substitution is performed **simultaneously** with parameter expansion, command substitution and arithmetic expansion.

The command list `<LIST>` is executed and its

- standard output filedescriptor in the `<( … )` form or
- standard input filedescriptor in the `>( … )` form

is connected to a FIFO or a file in `/dev/fd/`. The filename (where the filedescriptor is connected) is then used as a substitution for the `<(…)` -construct.

That, for example, allows to give data to a command that can't be reached by pipelining (that doesn't expect its data from `stdin` but from a file).

Scope

Note: According to multiple comments and sources, the scope of process substitution file descriptors is **not** stable, guaranteed, or specified by bash. Newer versions of bash (5.0+) seem to have shorter scope, and substitutions scope seems to be shorter than function scope. See stackexchange (https://unix.stackexchange.com/questions/425456/conditional-process-substitution) and stackoverflow (https://stackoverflow.com/questions/46660020/bash-what-is-the-scope-of-the-process-substitution); the latter discussion contains a script that can test the scoping behavior case-by-case
If a process substitution is expanded as an argument to a function, expanded to an environment variable during calling of a function, or expanded to any assignment within a function, the process substitution will be "held open" for use by any command within the function or its callees, until the function in which it was set returns. If the same variable is set again within a callee, unless the new variable is local, the previous process substitution is closed and will be unavailable to the caller when the callee returns.

In essence, process substitutions expanded to variables within functions remain open until the function in which the process substitution occured returns - even when assigned to locals that were set by a function's caller. Dynamic scope doesn't protect them from closing.

Examples

This code is useless, but it demonstrates how it works:

```
$ echo <(ls)
/dev/fd/63
```

The **output** of the `ls` -program can then be accessed by reading the file `/dev/fd/63` .

Consider the following:

```
diff <(ls "$first_directory") <(ls "$second_directory")
```

This will compare the contents of each directory. In this command, each *process* is *substituted* for a *file*, and diff doesn't see <(bla), it sees two files, so the effective command is something like

```
diff /dev/fd/63 /dev/fd/64
```

where those files are written to and destroyed automatically.

Avoiding subshells

> See Also: BashFAQ/024 (http://mywiki.wooledge.org/BashFAQ/024) – *I set variables in a loop that's in a pipeline. Why do they disappear after the loop terminates? Or, why can't I pipe data to read?*

One of the most common uses for process substitutions is to avoid the final subshell that results from executing a pipeline. The following is a **wrong** piece of code to count all files in `/etc` is:

```
counter=0

find /etc -print0 | while IFS= read -rd '' _; do
    ((counter++))
done

echo "$counter files" # prints "0 files"
```

Due to the pipe, the `while read; do … done` part is executed in a subshell (in Bash, by default), which means `counter` is only incremented within the subshell. When the pipeline finishes, the subshell is terminated, and the `counter` visible to `echo` is still at "0"!

Process substitution helps us avoid the pipe operator (the reason for the subshell):

```
counter=0

while IFS= read -rN1 _; do
    ((counter++))
done < <(find /etc -printf ' ')

echo "$counter files"
```

This is the normal input file redirection `< FILE`, just that the `FILE` in this case is the result of process substitution. It's important to note that the space is required in order to disambiguate the syntax from here documents.

```
: < <(COMMAND) # Good.
: <<(...) # Wrong. Will be parsed as a heredoc. Bash fails when it co
mes across the unquoted metacharacter ''('' 
: ><(...) # Technically valid but pointless syntax. Bash opens the pi
pe for writing, while the commands within the process substitution ha
ve their stdout connected to the pipe.
```

Process substitution assigned to a parameter

This example demonstrates how process substitutions can be made to resemble "passable" objects. This results in converting the output of `f` 's argument to uppercase.

```
f() {
    cat "$1" >"$x"
}

x=>(tr '[:lower:]' '[:upper:]') f <(echo 'hi there')
```

See the above section on scope

Bugs and Portability Considerations

- Process substitution is not specified by POSIX.
- Process substitution is disabled completely in Bash POSIX mode.
- Process substitution is implemented by Bash, Zsh, Ksh{88,93}, but not (yet) pdksh derivatives (mksh). Coprocesses may be used instead.
- Process substitution is supported only on systems that support either named pipes (FIFO - a special file) or the `/dev/fd/*` method for accessing open files. If the system doesn't support `/dev/fd/*`, Bash falls back to creating named pipes. Note that not all shells that support process substitution have that fallback.
- Bash evaluates process substitutions within array indices, but not other arithmetic contexts. Ksh and Zsh do not. (Possible Bug)

```
# print "moo"
dev=fd=1 _[1<(echo moo >&2)]=
# fork bomb
${dev[${dev='dev[1>(${dev[dev]})]'}]}
```

- Issues with wait, race conditions, etc: https://groups.google.com/forum/?fromgroups=#!topic/comp.unix.shell/GqLNzUA4ulA (https://groups.google.com/forum/?fromgroups=#!topic/comp.unix.shell/GqLNzUA4ulA)

See also

- Internal: Introduction to expansion and substitution
- Internal: Bash in the process tree (subshells)
- Internal: Redirection

Discussion

liungkejin, 2013/05/31 11:46 (), 2013/06/22 11:58 ()

I found a fun thing:

```
(echo "YES")> >(read str; echo "1:${str}:first";)> >(read sstr; e
cho "2:$sstr:two")> >(read ssstr; echo "3:$ssstr:three")
```

gives

```
1:2:3:YES:three:two:first
```