

# The coproc keyword

## Synopsis

```
coproc [NAME] command [redirections]
```

## Description

Bash 4.0 introduced *coprocesses*, a feature certainly familiar to ksh users. The `coproc` keyword starts a command as a background job, setting up pipes connected to both its stdin and stdout so that you can interact with it bidirectionally. Optionally, the co-process can have a name `NAME`. If `NAME` is given, the command that follows **must be a compound command**. If no `NAME` is given, then the command can be either simple or compound.

The process ID of the shell spawned to execute the coprocess is available through the value of the variable named by `NAME` followed by a `_PID` suffix. For example, the variable name used to store the PID of a coproc started with no `NAME` given would be `COPROC_PID` (because `COPROC` is the default `NAME`). The `wait` builtin command may be used to wait for the coprocess to terminate. Additionally, coprocesses may be manipulated through their `jobspec`.

## Return status

The return status of a coprocess is the exit status of its command.

## Redirections

The optional redirections are applied after the pipes have been set up. Some examples:

```
# redirecting stderr in the pipe
$ coproc { ls thisfiledoesntexist; read; } 2>&1
[2] 23084
$ IFS= read -ru ${COPROC[0]} x; printf '%s\n' "$x"
ls: cannot access thisfiledoesntexist: No such file or directory
```

```
#let the output of the coprocess go to stdout
$ { coproc mycoproc { awk '{print "foo" $0;fflush()}'; } >&3; } 3>&1
[2] 23092
$ echo bar >&${mycoproc[1]}
$ foobar
```

Here we need to save the previous file descriptor of stdout, because by the time we redirect the fds of the coprocess, stdout has already been redirected to the pipe.

# Pitfalls

## Avoid the final pipeline subshell

The traditional Ksh workaround to avoid the subshell when doing `command | while read` is to use a coprocess. Unfortunately, Bash's behavior differs.

In Ksh you would do:

```
# ksh93 or mksh/pdksh derivatives
ls |& # start a coprocess
while IFS= read -rp file; do print -r -- "$file"; done # read its output
```

In bash:

```
#DOESN'T WORK
$ coproc ls
[1] 23232
$ while IFS= read -ru ${COPROC[0]} line; do printf '%s\n' "$line"; done
bash: read: line: invalid file descriptor specification
[1]+  Done                  coproc COPROC ls
```

By the time we start reading from the output of the coprocess, the file descriptor has been closed.

See this FAQ entry on Greg's wiki (<http://mywiki.wooledge.org/BashFAQ/024>) for other pipeline subshell workarounds.

## Buffering

In the first example, we GNU awk's `fflush()` command. As always, when you use pipes the I/O operations are buffered. Let's see what happens with `sed` :

```
$ coproc sed s/^/foo/
[1] 22981
$ echo bar >&${COPROC[1]}
$ read -t 3 -ru ${COPROC[0]} _; (( $? > 127 )) && echo "nothing read"
nothing read
```

Even though this example is the same as the first `awk` example, the `read` doesn't return because the output is waiting in a buffer.

See this faq entry on Greg's wiki (<http://mywiki.wooledge.org/BashFAQ/009>) for some workarounds and more information on buffering issues.

## background processes

A coprocess' file descriptors are accessible only to the process from which the `coproc` was started. They are not inherited by subshells.

Here is a not-so-meaningful illustration. Suppose we want to continuously read the output of a coprocess and `echo` the result:

```
#NOT WORKING
$ coproc awk '{print "foo" $0;fflush()}'
[2] 23100
$ while IFS= read -ru ${COPROC[0]} x; do printf '%s\n' "$x"; done &
[3] 23104
bash: line 243: read: 61: invalid file descriptor: Bad file descriptor
```

This fails because the file descriptors created by the parent are not available to the subshell created by `&`.

A possible workaround:

```
#WARNING: for illustration purpose ONLY
# this is not the way to make the coprocess print its output
# to stdout, see the redirections above.
$ coproc awk '{print "foo" $0;fflush()}'
[2] 23109
$ exec 3<&${COPROC[0]}
$ while IFS= read -ru 3 x; do printf '%s\n' "$x"; done &
[3] 23110
$ echo bar >&${COPROC[1]}
$ foobar
```

Here, fd 3 is inherited.

## Examples

### Anonymous Coprocess

Unlike ksh, Bash doesn't have true anonymous coprocesses. Instead, Bash assigns FDs to a default array named `COPROC` if no `NAME` is supplied. Here's an example:

```
$ coproc awk '{print "foo" $0;fflush()}'
[1] 22978
```

This command starts in the background, and `coproc` returns immediately. Two new file descriptors are now available via the `COPROC` array. We can send data to our command:

```
$ echo bar >&${COPROC[1]}
```

And then read its output:

```
$ IFS= read -ru ${COPROC[0]} x; printf '%s\n' "$x"
foobar
```

When we don't need our command anymore, we can kill it via its pid:

```
$ kill ${COPROC_PID}
$
[1]+  Terminated                  coproc COPROC awk '{print "foo" $0;fflush()}'
```

### Named Coprocess

Using a named coprocess is simple. We just need a compound command (like when defining a function), and the resulting FDs will be assigned to the indexed array `NAME` we supply instead.

```
$ coproc mycoproc { awk '{print "foo" $0;fflush()}' ;}
[1] 23058
$ echo bar >&${mycoproc[1]}
$ IFS= read -ru ${mycoproc[0]} x; printf '%s\n' "$x"
foobar
$ kill $mycoproc_PID
$
[1]+  Terminated                  coproc mycoproc { awk '{print "foo" $0;fflush
()}' ; }
```

## Redirecting the output of a script to a file and to the screen

```
#!/bin/bash
# we start tee in the background
# redirecting its output to the stdout of the script
{ coproc tee { tee logfile ;} >&3 ;} 3>&1
# we redirect stding and stdout of the script to our coprocess
exec >&${tee[1]} 2>&1
```

## Portability considerations

- The `coproc` keyword is not specified by POSIX(R)
- The `coproc` keyword appeared in Bash version 4.0-alpha
- The `-p` option to Bash's `print` loadable is a NOOP and not connected to Bash coprocesses in any way. It is only recognized as an option for ksh compatibility, and has no effect.
- The `-p` option to Bash's `read` builtin conflicts with that of all kshes and zsh. The equivalent in those shells is to add a `\?prompt` suffix to the first variable name argument to `read` . i.e., if the first variable name given contains a `?` character, the remainder of the argument is used as the prompt string. Since this feature is pointless and redundant, I suggest not using it in either shell. Simply precede the `read` command with a `printf %s prompt >&2` .

## Other shells

ksh93, mksh, zsh, and Bash all support something called "coprocesses" which all do approximately the same thing. ksh93 and mksh have virtually identical syntax and semantics for coprocs. A *list* operator: `|&` is added to the language which runs the preceding *pipeline* as a coprocess (This is another reason not to use the special `|&` pipe operator in Bash – its syntax is conflicting). The `-p` option to the `read` and `print` builtins can then be used to read and write to the pipe of the coprocess (whose FD isn't yet known). Special redirects are added to move the last spawned coprocess to a different FD: `<&p` and `>&p` , at which point it can be accessed at the new FD using ordinary redirection, and another coprocess may then be started, again using `|&` .

zsh coprocesses are very similar to ksh except in the way they are started. zsh adds the shell reserved word `coproc` to the pipeline syntax (similar to the way Bash's `time` keyword works), so that the pipeline that follows is started as a coproc. The coproc's input and output FDs can then be accessed and moved using the same `read / print -p` and redirects used by the ksh shells.

It is unfortunate that Bash chose to go against existing practice in their coproc implementation, especially considering it was the last of the major shells to incorporate this feature. However, Bash's method accomplishes the same without requiring nearly as much additional syntax. The `coproc`

keyword is easy enough to wrap in a function such that it takes Bash code as an ordinary argument and/or stdin like `eval`. Coprocess functionality in other shells can be similarly wrapped to create a `COPROC` array automatically.

## Only one coprocess at a time

The title says it all, complain to the bug-bash mailing list if you want more. See <http://lists.gnu.org/archive/html/bug-bash/2011-04/msg00056.html> (<http://lists.gnu.org/archive/html/bug-bash/2011-04/msg00056.html>) for more details

The ability to use multiple coprocesses in Bash is considered "experimental". Bash will throw an error if you attempt to start more than one. This may be overridden at compile-time with the `MULTIPLE_COPROCS` option. However, at this time there are still issues – see the above mailing list discussion.

## See also

- Anthony Thyssen's Coprocess Hints (<http://www.ict.griffith.edu.au/anthony/info/shell/coprocesses.hints>) - excellent summary of everything around the topic

## Discussion

Anthony Thyssen (<http://www.cit.griffith.edu.au/~anthony/>), 2010/06/18 02:18 ()

Can you do a coprocess using only regular shell file descriptors? or perhaps using named pipes.

```
mknod ls_output ls > ls_output &
```

```
while read line; do
```

```
...
```

```
done < ls_output
```

Jan Schampera, 2010/06/18 08:36 ()

Depending on your specific needs there is of course the possibility to use

- named pipes
- process substitution
- command substitution
- ...

Coprocesses as described here are just a very easy to use solution for those tasks. It's simple to setup, use and terminate a coprocess.

Anthony Thyssen (<http://www.ict.griffith.edu.au/anthony/>), 2011/09/28 07:30 ()

Seems to me it is about just as complex as using temporary named pipes.

Of course most difficulties with handling a coprocess is often the handling of the data streams especially if you want more than just stdin/stdout.

Since my initial feedback I have written what I would hope is the start of a guide to using co-processes. And yes it can be very much worth the effort.

<http://www.ict.griffith.edu.au/anthony/info/shell/co-processes.hints>

(<http://www.ict.griffith.edu.au/anthony/info/shell/co-processes.hints>) Feedback welcome  
Anthony Thyssen [A.Thyssen@griffith.edu.au](mailto:A.Thyssen@griffith.edu.au) (<mailto:A.Thyssen@griffith.edu.au>)

Jan Schampera, [2011/09/28 20:08.\(\)](#)

Wow... excellent knowledge collection. I crosslinked it here

Adam Baxter (<http://blog.voltage.org>), [2016/06/03 08:27.\(\)](#)

Seems to have disappeared. It's archived at

<https://web.archive.org/web/20151221031303/http://www.ict.griffith.edu.au/anthony/info/shell/co-processes.hints>

(<https://web.archive.org/web/20151221031303/http://www.ict.griffith.edu.au/anthony/info/shell/co-processes.hints>)

SZABÓ Gergely (<http://linux.subogero.com>), [2012/05/10 08:20.\(\)](#), [2012/07/01 11:48.\(\)](#)

I've found an easier way to read from the coprocess. In the example below `szg` is an interactive calculator capable of converting hex numbers.

```
$ coproc S { szg; }
[1] 1234
$ echo XffffD >&${S[1]}
$ head -n 1 <&${S[0]}
65535
```

George Caswell (<http://scope-eye.net>), [2016/04/13 00:25.\(\)](#)

You have to be careful of what you attach to file descriptors in the shell. When programs are using buffered I/O they may read more data than they actually need. For instance:

```
coproc S { while read line; do echo "$line"; echo "Cool, huh?"; done; }
```

```
echo "Test" >&${S[1]}
```

```
head -1 <&${S[0]}
```

```
Test
```

```
echo "Test2" >&${S[1]}
```

```
head -1 <&${S[0]}
```

```
Test2
```



Notice that "head" never prints the "Cool, huh?" line? That's because internally "head" is doing something like this:

(during a call to "getline()" or "fgets()" or whatever...):

- Read in **as much data as I presently can** into my buffer.
- Scan buffer for the first newline character.
- Cut the string from the start of the buffer to the first newline character out of the buffer. Return it.

That aggressive buffering is done to minimize the time spent doing actual I/O. But it means there's a good chance the process will consume more than a single line of input from that file descriptor.

Basically you have to put one process in charge of reading lines out of that file descriptor. The shell can be that process, or you can delegate the task to another process. But most programs aren't written with the assumption that they're sharing their input file.

 syntax/keywords/coproc.txt  Last modified: 2013/04/14 12:36 by thebonsai

---

This site is supported by Performing Databases - your experts for database administration

---

Bash Hackers Wiki

---



Except where otherwise noted, content on this wiki is licensed under the following license:  
GNU Free Documentation License 1.3