How to make a pipe loop in bash

Asked 16 years, 3 months ago Modified 9 years, 9 months ago

Viewed 22k times

23

Assume that I have programs `P0`, `P1`, ...`P(n-1)` for some `n > 0`. How can I easily redirect the output of program `Pi` to program `P(i+1 mod n)` for all `i` (`0 <= i < n`)?

For example, let's say I have a program `square`, which repeatedly reads a number and than prints the square of that number, and a program `calc`, which sometimes prints a number after which it expects to be able to read the square of it. How do I connect these programs such that whenever `calc` prints a number, `square` squares it returns it to `calc`?

Edit: I should probably clarify what I mean with "easily". The named pipe/fifo solution is one that indeed works (and I have used in the past), but it actually requires quite a bit of work to do properly if you compare it with using a bash pipe. (You need to get a not yet existing filename, make a pipe with that name, run the "pipe loop", clean up the named pipe.) Imagine you could no longer write `prog1 | prog2` and would always have to use named pipes to connect programs.

I'm looking for something that is almost as easy as writing a "normal" pipe. For instance something like `{ prog1 |`

`prog2 } >&0` would be great.

bash

Share

Improve this question

Follow

edited Sep 4, 2008 at 7:38

asked Sep 2, 2008 at 18:40

mweerden

14k ● 5 ● 34 ● 32

7 Answers

Sorted by: Highest score (default)

27

After spending quite some time yesterday trying to redirect `stdout` to `stdin`, I ended up with the following method. It isn't really nice, but I think I prefer it over the named pipe/fifo solution.

```
read | { P0 | ... | P(n-1); } >/dev/fd/0
```

The `{ ... } >/dev/fd/0` is to redirect stdout to stdin for the pipe sequence as a whole (i.e. it redirects the output of P(n-1) to the input of P0). Using `>&0` or something similar does not work; this is probably because bash assumes `0` is read-only while it doesn't mind writing to `/dev/fd/0`.

The initial `read`-pipe is necessary because without it both the input and output file descriptor are the same pts device (at least on my system) and the redirect has no effect. (The pts device doesn't work as a pipe; writing to it puts things on your screen.) By making the input of the `{ ... }` a normal pipe, the redirect has the desired effect.

To illustrate with my `calc`/`square` example:

```
function calc() {
  # calculate sum of squares of numbers 0,..,10

  sum=0
  for ((i=0; i<10; i++)); do
    echo $i                   # "request" the square o

    read ii                   # read the square of i
    echo "got $ii" >&2          # debug message

    let sum=$sum+$ii
  done

  echo "sum $sum" >&2           # output result to std
}

function square() {
  # square numbers

  read j                         # receive first "requ
  while [ "$j" != "" ]; do
    let jj=$j*$j
    echo "square($j) = $jj" >&2  # debug message

    echo $jj                     # send square

    read j                       # receive next "reque
  done
}
```

```
read | { calc | square; } >/dev/fd/0
```

Running the above code gives the following output:

```
square(0) = 0
got 0
square(1) = 1
got 1
square(2) = 4
got 4
square(3) = 9
got 9
square(4) = 16
got 16
square(5) = 25
got 25
square(6) = 36
got 36
square(7) = 49
got 49
square(8) = 64
got 64
square(9) = 81
got 81
sum 285
```

Of course, this method is quite a bit of a hack. Especially the `read` part has an undesired side-effect: termination of the "real" pipe loop does not lead to termination of the whole. I couldn't think of anything better than `read` as it seems that you can only determine that the pipe loop has terminated by try to writing write something to it.

Share Improve this answer

Follow

answered Sep 4, 2008 at 8:22

mweerden

14k ● 5 ● 34 ● 32

1 Nice solution. I had to do something similar using netcat inside a loop and worked around the 'read' side effect by 'closing' its input with an 'echo'. In the end it was something like this : echo | read | { P0 | ... | P(n-1); } >/dev/fd/0
– Thiago Padilha Nov 30, 2011 at 16:29

2 Instead of `echo|read` one could use a command that terminates immediately, like `:` (`true`), e.g. `: | { cmd | cmd >/dev/fd/0}`. Example: `: | { nc -lp 5000 >/dev/fd/0; }` is a simple echo server that correctly terminates on client EOF. – regnarg Jul 31, 2014 at 20:46

15

A named pipe might do it:

```
$ mkfifo outside
$ <outside calc | square >outside &
$ echo "1" >outside ## Trigger the loop to start
```

Share Improve this answer

Follow

answered Sep 2, 2008 at 20:57

Douglas Leeder

53.3k ● 9 ● 99 ● 138

Could you explain the line "<outside calc | square >outside &"? I am unsure about <outside and >outside.
– Léo Léopold Hertz 준영 May 7, 2009 at 18:35

They're standard shell redirects - reading in from 'outside' and outputting to 'outside'. outside is a fifo, so everything written to it, comes out the read side. – Douglas Leeder May 8, 2009 at 8:29

I tried this code, but it doesn't work. It seems that this line: `<outside calc | square >outside &` finishes immediately. – RnMss Jun 19, 2013 at 10:03

@RnMss maybe `awk` doesn't wait for input if its stdin isn't a terminal? I suggest you ask a new question, explaining why you need to do a loop of input with awk, and what you've already tried. – Douglas Leeder Jun 19, 2013 at 12:48

There's also a solution without named pipe script - Bash: create anonymous fifo - Super User – user202729 Aug 7, 2022 at 17:43

5

This is a very interesting question. I (vaguely) remember an assignment very similar in college 17 years ago. We had to create an array of pipes, where our code would get filehandles for the input/output of each pipe. Then the code would fork and close the unused filehandles.

I'm thinking you could do something similar with named pipes in bash. Use mknod or mkfifo to create a set of pipes with unique names you can reference then fork your program.

Share Improve this answer Follow

answered Sep 2, 2008 at 19:16

Mark Witczak
1,613 ● 2 ● 15 ● 13

3

My solutions uses pipexec (Most of the function implementation comes from your answer):

square.sh

```
function square() {
  # square numbers

  read j                         # receive first "requ
  while [ "$j" != "" ]; do
    let jj=$j*$j
    echo "square($j) = $jj" >&2  # debug message

    echo $jj                     # send square

    read j                       # receive next "reque
  done
}

square $@
```

calc.sh

```
function calc() {
  # calculate sum of squares of numbers 0,..,10

  sum=0
  for ((i=0; i<10; i++)); do
    echo $i                    # "request" the square o

    read ii                    # read the square of i
    echo "got $ii" >&2           # debug message

    let sum=$sum+$ii
 done

 echo "sum $sum" >&2           # output result to stde
}
```

```
calc $@
```

The command

```
pipexec [ CALC /bin/bash calc.sh ] [ SQUARE /bin/bash
    "{CALC:1>SQUARE:0}" "{SQUARE:1>CALC:0}"
```

The output (same as in your answer)

```
square(0) = 0
got 0
square(1) = 1
got 1
square(2) = 4
got 4
square(3) = 9
got 9
square(4) = 16
got 16
square(5) = 25
got 25
square(6) = 36
got 36
square(7) = 49
got 49
square(8) = 64
got 64
square(9) = 81
got 81
sum 285
```

Comment: pipexec was designed to start processes and build arbitrary pipes in between. Because bash functions cannot be handled as processes, there is the need to have the functions in separate files and use a separate bash.

Share Improve this answer Follow

answered Mar 14, 2015 at 20:30

Andreas Florath
4,612 ● 24 ● 34

1

Named pipes.

Create a series of fifos, using mkfifo

i.e fifo0, fifo1

Then attach each process in term to the pipes you want:

processn < fifo(n-1) > fifon

Share Improve this answer Follow

answered Sep 2, 2008 at 20:57

1729
5,031 ● 4 ● 29 ● 17

-1

I doubt sh/bash can do it. ZSH would be a better bet, with its MULTIOS and coproc features.

Share Improve this answer Follow

answered Sep 2, 2008 at 20:31

Penz
5,608 ● 5 ● 33 ● 30

1 Could you give an example about Zsh? I am interested in it.
– Léo Léopold Hertz 준영 May 7, 2009 at 18:36

-2

A command stack can be composed as string from an array of arbitrary commands and evaluated with eval. The following example gives the result 65536.

```
function square ()
{
  read n
  echo $((n*n))
}    # ----------  end of function square  ----------

declare -a  commands=( 'echo 4' 'square' 'square' 'squ

#-----------------------------------------------------
-
#   build the command stack using pipes
#-----------------------------------------------------
-
declare     stack=${commands[0]}

for (( COUNTER=1; COUNTER<${#commands[@]}; COUNTER++ )
  stack="${stack} | ${commands[${COUNTER}]}"
done

#-----------------------------------------------------
-
#   run the command stack
#-----------------------------------------------------
-
eval "$stack"
```

Share Improve this answer

Follow

answered Jan 21, 2009 at 9:56

Fritz G. Mehner
17.2k ● 3 ● 35 ● 43

1 I don't think you're answering the question. – reinierpost Jan 29, 2010 at 15:04