Bash Pipe Handling

Asked 16 years, 4 months ago Modified 1 year, 7 months ago Viewed 13k times

31

Does anyone know how bash handles sending data through pipes?

```
cat file.txt | tail -20
```

Does this command print all the contents of file.txt into a buffer, which is then read by tail? Or does this command, say, print the contents of file.txt line by line, and then pause at each line for tail to process, and then ask for more data?

The reason I ask is that I'm writing a program on an embedded device that basically performs a sequence of operations on some chunk of data, where the output of one operation is send off as the input of the next operation. I would like to know how linux (bash) handles this so please give me a general answer, not specifically what happens when I run "cat file.txt | tail -20".

EDIT: Shog9 pointed out a relevant Wikipedia Article, this didn't lead me directly to the article but it helped me find this: http://en.wikipedia.org/wiki/Pipeline_%28Unix%29#Implementation which did have the information I was looking for.

I'm sorry for not making myself clear. Of course you're using a pipe and of course you're using stdin and stdout of the respective parts of the command. I had assumed that was too obvious to state.

What I'm asking is how this is handled/implemented. Since both programs cannot run at once, how is data sent from stdin to stdout? What happens if the first program generates data significantly faster than the second program? Does the system just run the first command until either it's terminated or it's stdout buffer is full, and then move on to the next program, and so on in a loop until no more data is left to be processed or is there a more complicated mechanism?

linux bash pipe

Share
Improve this question
Follow

edited May 10, 2023 at 9:44
Jonas
128k ● 100 ● 326 ● 405

asked Aug 21, 2008 at 0:20
num1
4,955 ● 4 ● 34 ● 50

3 Answers

Sorted by: Highest score (default)

55

I decided to write a slightly more detailed explanation.

The "magic" here lies in the operating system. Both programs do start up at roughly the same time, and run at the same time (the operating system assigns them slices of time on the processor to run) as every other simultaneously running process on your computer (including the terminal application and the kernel). So, before any data gets passed, the processes are doing whatever initialization necessary. In your example, tail is parsing the '-20' argument and cat is parsing the 'file.txt' argument and opening the file. At some point tail will get to the point where it needs input and it will tell the operating system that it is waiting for input. At some other point (either before or after, it doesn't matter) cat will start passing data to the operating system using stdout. This goes into a buffer in the operating system. The next time tail gets a time slice on the processor after some data has been put into the buffer by cat, it will retrieve some amount of that data (or all of it) which leaves the buffer on the operating system. When the buffer is empty, at some point tail will have to wait for cat to output more data. If cat is outputting data much faster than tail is handling it, the buffer will expand. cat will eventually be done outputting data, but tail will still be processing, so cat will close and tail will process all remaining data in the buffer. The operating system will signal tail when their is no more incoming data with an EOF. Tail will process the remaining data. In this case, tail is probably just receiving all the data into a circular buffer of 20 lines, and when it is signalled by the operating system that there is no more incoming data, it then dumps the last twenty lines to its own stdout, which just gets displayed in the terminal. Since tail is a much simpler program than cat, it will likely spend most of the time waiting for cat to put data into the buffer.

On a system with multiple processors, the two programs will not just be sharing alternating time slices on the same processor core, but likely running at the same time on separate cores.

To get into a little more detail, if you open some kind of process monitor (operating system specific) like 'top' in Linux you will see a whole list of running processes, most of which are effectively using 0% of the processor. Most applications, unless they are crunching data, spend most of their time doing nothing. This is good, because it allows other processes to have unfettered access to the processor according to their needs. This is accomplished in basically three ways. A process could get to a sleep(n) style instruction where it basically tells the kernel to wait n milliseconds before giving it another time slice to work with. Most commonly a program needs to wait for something from another program, like 'tail' waiting for more data to enter the buffer. In this case the operating system will wake up the process when more data is available. Lastly, the kernel can preempt a process in the middle of execution, giving some processor time slices to other processes. 'cat' and 'tail' are simple programs. In this

example, tail spends most of it's time waiting for more data on the buffer, and cat spends most of it's time waiting for the operating system to retrieve data from the harddrive. The bottleneck is the speed (or slowness) of the physical medium that the file is stored on. That perceptible delay you might detect when you run this command for the first time is the time it takes for the read heads on the disk drive to seek to the position on the harddrive where 'file.txt' is. If you run the command a second time, the operating system will likely have the contents of file.txt cached in memory, and you will not likely see any perceptible delay (unless file.txt is very large, or the file is no longer cached.)

Most operations you do on your computer are IO bound, which is to say that you are usually waiting for data to come from your harddrive, or from a network device, etc.

Share Improve this answer Follow

answered Aug 21, 2008 at 5:46

postfuturist

22.6k ● 11 ● 66 ● 85

This incorrectly implies that the output buffer from `cat` will be grown indefinitely; but the OS can also block the writing when the output buffer reaches its maximum size. – tripleee Dec 22, 2019 at 11:03

1

Shog9 already referenced the Wikipedia article, but the implementation section has the details you want. The basic implementation is a bounded buffer.

Share Improve this answer Follow

answered Aug 21, 2008 at 1:45

David Schlosnagle

4,323 ● 2 ● 24 ● 16

0

cat will just print the data to standard out, which happens to be redirected to the standard in of tail. This can be seen in the man page of bash.

In other words, there is no pausing going on, tail is just reading from standard in and cat is just writing to standard out.

Share Improve this answer Follow

answered Aug 21, 2008 at 0:29

Mike Stone

44.6k ● 30 ● 114 ● 140