PETER SOBOT

AUGUST 7, 2013

Pipes and Filters

Pipelines are an extremely useful (and surprisingly underused) architectural pattern in modern software engineering. The concept of using pipes and filters to control the flow of data through software has been around since the 1970s, when the first Unix shells were created. If you've ever used the pipe ("`|`") character in a terminal emulator, you've made use of the pipe-and-filter idiom. Take the following example:

```
cat /usr/share/dict/words |     # Read in the system's dictionary.
grep purple |                   # Find words containing 'purple'
awk '{print length($1), $1}' |  # Count the letters in each word
sort -n |                       # Sort lines ("${length} ${word}")
tail -n 1 |                     # Take the last line of the input
cut -d " " -f 2 |               # Take the second part of each line
cowsay -f tux                   # Put the resulting word into Tux's
mouth
```

When run with `bash`, this pipeline returns a charming little ASCII art version of Tux, the Linux penguin, saying the longest word in the dictionary that contains the word "purple":

```
 _____________
< unimpurpled >
 -------------
    \
     \
         .--.
        |o_o |
        |:_/ |
       //   \ \
      (|     | )
     /'\_   _/`\
     \___)=(___/
```

The Life Cycle of a Pipeline

This little bit of code actually does quite a lot as soon as it's executed. The moment you hit enter, the following steps occur:

1. Seven (seven!) processes are immediately spawned by the shell.
2. The standard input (`stdin`) and standard output (`stdout`) file descriptors of each process are redirected to the shell's internal buffers. (Each of these buffers is 512 bytes long on my machine, as measured by running `ulimit -a`.)
3. The source process, `cat`, starts to read from its file and output to its `stdout`. This data flows through the first pipe into the first buffer, quickly hitting the buffer size limit imposed by `bash`. As soon as this limit is reached, `cat` is blocked in its `write(2)` call. This is where pipelines really shine: the execution of `cat` is **implicitly** paused by the pipeline's inability to handle more data. (For those familiar with the concept of **coroutines**, each process here is acting as a sort of coroutine.)
4. The first filtering process, `grep`, starts with a `read(2)` call on its `stdin` pipe. When the process first spawns, the pipe is empty - so the entire process **blocks** until it can read more data. Again, we see *implicit* execution control based on the availability of data. As soon as the preceding command in the pipeline fills or flushes its buffer, `grep`'s `read(2)` call returns and it can filter the lines that it's read in from the preceding process. Every line that matches the provided pattern is immediately printed to `grep`'s `stdout`, which will be available to the next program in the pipeline.
5. `awk` functions exactly like `grep` does, only on different buffers. When data is available, `awk` resumes execution and processes the data, incrementally writing its results to its `stdout`. When data is not available, `awk` is blocked and unable to run.

6. `sort` operates slightly differently than the preceding two processes. As a sorting operation must take place on the entirity of the data, `sort` maintains a buffer (on disk) of the entire input received thus far. (There's no point in providing a sorted list of all the data received so far, only to have it invalidated by a later piece of data.) As soon as `sort`'s `stdin` *closes*, `sort` can print its output to its `stdout`, as it knows no more data will be read.

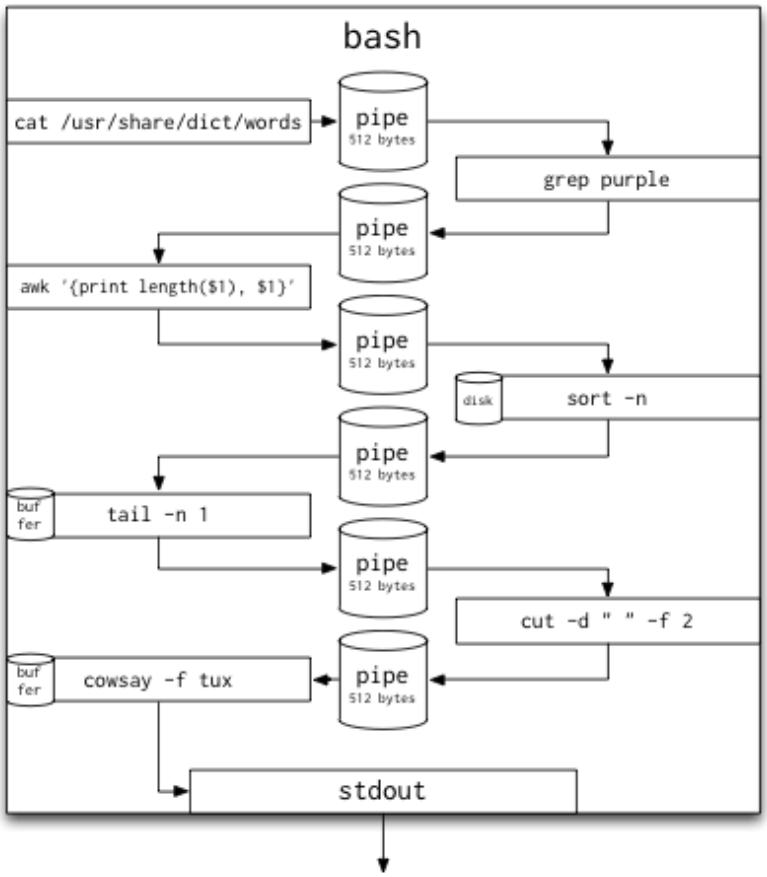
7. `tail` is somewhat similar to `sort` in that it cannot produce any data on `stdout` until the entirity of the data has been received. This invocation doesn't need to maintain a large internal buffer, as it only cares about the last line of the input.

8. `cut` operates as an incremental filter, just like `grep` and `ack` do.

9. `cowsay` operates just like `tail` does — waiting to receive the full input before processing it, as it must calculate metrics based on the length of the input data. (Printing properly aligned ASCII art is no easy task!)

Using a pipeline for this task should seem like a no-brainer. Every task being done here deals with *filtering* data. Existing data sets are changed at each step. Every process does its own job, and does it quite well, as the Unix philosophy recommends. Each process could be swapped out for another with very little effort.

If you wanted to visualize the pipeline that `bash` automatically sets up for you, it would look something like this:



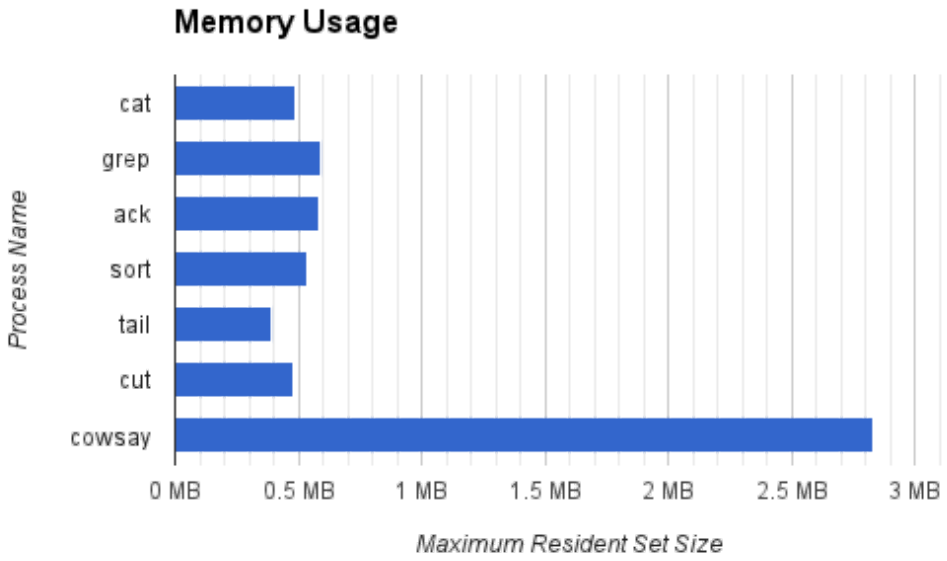
Performance and Complexity

One other advantage of pipelines is their **inherently good performance**. Let's use a modified version of the command we ran above to find the memory and CPU usage of each filter component in the pipeline.

```
/usr/bin/time -l cat /usr/share/dict/words 2> cat.time.txt |
/usr/bin/time -l grep purple 2> grep.time.txt |
/usr/bin/time -l awk '{print length($1), $1}' 2> awk.time.txt |
/usr/bin/time -l sort -n 2> sort.time.txt |
/usr/bin/time -l tail -n 1 2> tail.time.txt |
/usr/bin/time -l cut -d " " -f 2 2> cut.time.txt |
/usr/bin/time -l cowsay -f tux 2> cowsay.time.txt
```

(*Aside:* I'm calling `/usr/bin/time` here to avoid using my shell's built-in `time` command, which doesn't support the `-l` flag to print out detailed stats. If you're on Linux, you'll want to use the `-v` flag, which does the same thing. The `2> something.time.txt` syntax redirects `stderr` to a file, while leaving `stdout` pointing at a pipe.)

After running this command and checking the maximum resident set size, as well as the number of voluntary and involuntary context switches, we can start to see a couple very important things.



- The maximum amount of memory used by any one filter was 2,830,336 bytes, by `cowsay`, due to the fact that it's implemented in Perl. (Just spawning a Perl interpreter on my machine uses 1,126,400 bytes!) The minimum was 389,120 bytes, used by `tail`.
    - Even though our original source file (`/usr/share/dict/words`) was 2.4 MB in size, most of the filters in the pipeline don't even use one fifth of that amount of memory! Thanks to the fact that the pipeline **only stores what it can process** in memory, this solution is very memory-efficient and lightweight. Processing a file of any size would *not* have changed the memory usage of this solution - the pipeline runs in **effectively constant space**.
- Notice that the first two processes, `cat` and `grep`, have a large number of **voluntary context switches**. This is a fancy way of saying **blocking on IO**. `cat` must voluntarily context switch into the operating

system when reading the original file from disk, then again when writing to its `stdout` pipe. `grep` must voluntarily context switch when reading from its `stdin` pipe and writing to its `stdout`. The reason that `ack`, `sort`, `tail` and `cut` don't have as many context switches is that they deal with less data — `grep` has already filtered the data for them, resulting in only twelve lines that match the provided pattern. These twelve lines can fit easily within one pipe buffer.

- `cowsay` seems to have an unusually high number of *involuntary* context switches, which are probably caused by the process's time quantum expiring. I'm going to attribute that to the fact that it's written in Perl, and that it takes ~30 milliseconds of CPU time to run, compared to the immeasurably small time that the other programs take to run.

Note that although this example pipeline is amazingly simple, if any of these processes were doing complex computations, they could be *automatically parallelized* on multiple processors. Aren't pipelines awesome?

Errors

Yes indeed - pipelines are awesome. They make efficient use of memory and CPU time, have automatic and implicit execution scheduling based on data availability, and they're super easy to create. Why would you *not* want to use pipelines whenever possible?

The answer: **error handling**. If something goes wrong in one of the parts of the pipeline, the entire pipeline fails completely.

Let's try out this pipeline, with an added command that I've written in Python. `fail.py` echoes its standard input

to standard output, but has a 50% chance of crashing before reading a line.

```
cat /usr/share/dict/words |     # Read in the system's dictionary.
grep purple |                   # Find words containing 'purple'
awk '{print length($1), $1}' |  # Count the letters in each word
sort -n |                       # Sort lines ("${length} ${word}")
python fail.py |                # Play Russian Roulette with our data!
tail -n 1 |                     # Take the last line of the input
cut -d " " -f 2 |               # Take the second part of each line
cowsay -f tux                   # Put the resulting word into Tux's
mouth
```

The source of `fail.py`:

```
import sys
import random

while True:
    if random.choice([True, False]):
        sys.exit(1)
    line = sys.stdin.readline()
    if not line:
        break
    sys.stdout.write(line)
    sys.stdout.flush()
```

So, what happens in this case? When `fail.py` fails while reading the input, its `stdin` and `stdout` pipes close. This essentially **cuts the pipeline in half**. Let's take a look at what each process does as you get further and further away from the failed process.

1. `sort`, the process immediately before `python`, immediately receives a `SIGPIPE` signal to tell it that one of the pipes it has open (its `stdout`) has closed. It can choose to handle this `SIGPIPE` immediately, or can try to `write(2)` again - but that `write(2)` call will return `-1` anyways. No longer able to write its output anywhere, `sort` will exit, closing its *own* `stdin` pipe, causing the process preceding it to do the same thing. This *cascading shutdown* proceeds all the way up to the first process in the pipeline. (Of course, a process doesn't *have* to shut down

when it encounters a write error or receives a `SIGPIPE`, but these processes don't have any other behaviour if their output pipes close.)

2. `tail`, the process immediately after `python`, also receives a `SIGPIPE` signal as soon as its `stdin` pipe closes. It can choose to handle the `SIGPIPE` with a handler, or to ignore the signal, but either way - its next call to `read(2)` will return an error code. This event is *indistinguishable* from the end-of-stream event that `tail` receives anyways when the input stream is done. Hence, `tail` will interpret this as a normal end-of-stream event, and will behave as expected.

3. `cut` will also behave as expected when the stream closes.

4. `cowsay` will behave as expected when the stream closes, printing out the last word in the sorted list that was received *before* the `python` process crashed.

The result?

```
 __________
< repurple >
 ----------
   \
    \
        .--.
       |o_o |
       |:_/ |
      //   \ \
     (|     | )
    /'\_   _/`\
    \___)=(___/
```

Notice that Tux is no longer saying `unimpurpled`, the word that he was saying before. The word is wrong! The output of our command pipeline **is incorrect**. Although one of the filters crashed, we still got a response back - and all of the steps in the pipeline after the crashed filter still executed as expected.

What's worse - if we check the return code of the pipeline, we get:

```
bash-3.2$ echo $?
0
```

`bash` helpfully reports to us that the pipeline *executed correctly*. This is due to the fact that `bash` only reports the exit status of the *last* process in the pipeline. The only way to detect an issue earlier in the pipeline is to check `bash`'s relatively unknown `$PIPESTATUS` variable:

```
bash-3.2$ echo ${PIPESTATUS[*]}
0 0 0 0 1 0 0 0
```

This array stores the return codes of every process in the previous pipe chain - and only *here* we can see that one of the filters crashed.

This is one of the major drawbacks about using traditional UNIX pipes. Detecting an error while the pipeline is still processing data requires some form of **out-of-band** signalling to detect a failed process and send a message to the other processes. (This is easy to do when you have more than one input pipe to a filter, but becomes difficult if you're just using UNIX pipes.)

Other Uses for Pipelines

So, that's great. We can make a memory-efficient pipeline to create ASCII art penguins. I can hear the questions now:

> How are pipes useful in the real world?
>
> How could pipes help me in my web app?

Valid questions. Pipes work great when your data can be **divided into very small chunks** and when **processing can**

be done incrementally. Here's a couple examples.

Let's say you have a folder full of `.flac` files - very high quality music. You want to put these files on your MP3 player, but it doesn't support `.flac`. And for some reason, your computer doesn't have more than 10 megabytes of available RAM. Let's use a pipeline:

```
ls *.flac |
while read song
do
    flac -d "$song" --stdout |
    lame -V2 - "$song".mp3
done
```

This command is a little more complicated than the simple pipeline we used above. First, we're using a built in `bash` construct - the `while` loop with a `read` command inside. This reads every line of the input (which we're piping in from `ls`) and executes the inner code once per line. Then, the inner loop invokes `flac` to decode the song, and `lame` to encode the song to an MP3.

How memory-efficient is this pipeline? After running it on a folder full of 115MB of FLAC files, only **1.3MB** of memory was used.

Let's say you have a web app, and that you're using your favourite web framework to serve it. If a user submits a form to your app, you need to do some very expensive processing on the back-end. The form data needs to be sanitized, verified by calling an external API, and saved as a PDF file. All of this work can't happen in the web server

itself, as it would be too slow. (*Yes - this example is a little bit contrived, but isn't that far off from some use cases I've seen.*) Again, let's use a pipeline:

```
my_webserver |
line_sanitizer |
verifier |
pdf_renderer
```

Whenever a user submits a form to `my_webserver`, it can write a line of JSON to its `stdout`. Let's say this line looks like:

```
{"name": "Raymond Luxury Yacht", "organization": "Flying Circus"}
```

The next process in the pipeline, `line_sanitizer`, can then run some logic on each line:

```
import sys
import json

for line in sys.stdin:
    obj = json.loads(line)

    if "Eric Idle" in obj['name']:
        # Ignore forms submitted by Eric Idle.
        continue

    sys.stdout.write(line)
    sys.stdout.flush()
```

The next process can verify that the organization exists:

```
import sys
import json
import requests

for line in sys.stdin:
    obj = json.loads(line)
    org = obj['organization']
    resp = requests.get("http://does.it/exist", data=org)

    if resp.response_code == 404:
        continue
```

```
        sys.stdout.write(line)
        sys.stdout.flush()
```

Finally, the last process can bake any lines that remain into PDF files.

```
import sys
import json
import magical_pdf_writer_that_doesnt_exist as writer

for line in sys.stdin:
    obj = json.loads(line)
    writer.write_to_file(obj)
```

And there you have it - an *asynchronous*, extremely-memory-efficient pipeline that can process huge amounts of data, with a very, very small amount of code.

One question remains in this example, however. How do we handle errors that might occur? If Eric Idle submits a form to our website, and we decide to reject the form, how do we notify him? One very UNIX-y way of doing so would be to create a **named pipe** that handles all failed requests:

```
mkfifo errors  # create a named pipe for our errors

my_webserver |
line_sanitizer 2> errors |
verifier 2> errors |
pdf_renderer 2> errors
```

Any process could read from our own custom “errors” pipe, and each process in the pipeline would output its failed inputs into that pipe. We could attach a reader to that pipe that sends out emails on failure:

```
mkfifo errors              # create a named pipe for our errors
email_on_error < errors &  # add a reader to this pipe

my_webserver |
line_sanitizer 2> errors |
verifier 2> errors |
pdf_renderer 2> errors
```

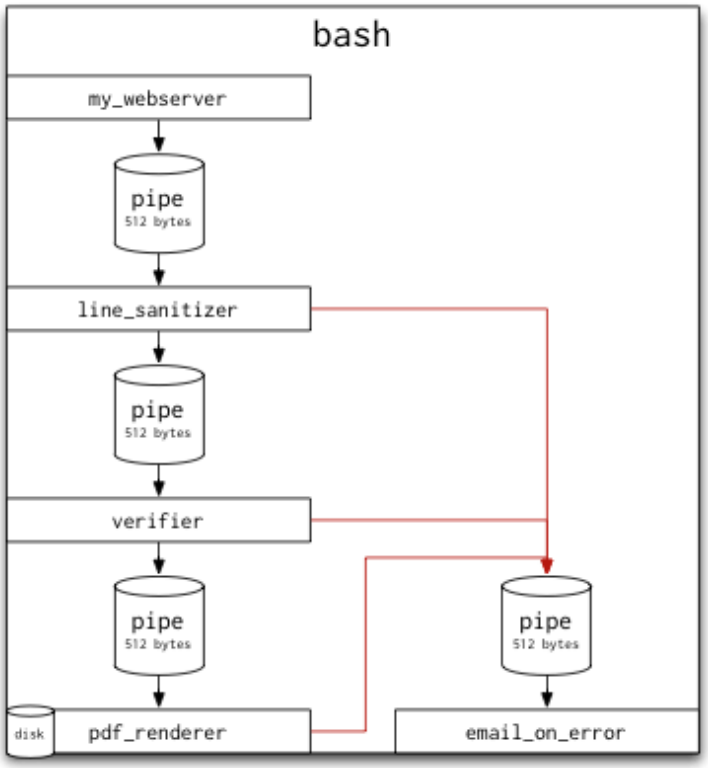
Then, if our `line_sanitizer` wanted to reject a line, its behaviour would look like this:

```
import sys
import json

for line in sys.stdin:
    obj = json.loads(line)

    if "Eric Idle" in obj['name']:
        sys.stderr.write(line)
        sys.stderr.flush()
    else:
        sys.stdout.write(line)
        sys.stdout.flush()
```

This pipeline would look a little bit different. (Red lines represent `stderr` output.)



Distributed Pipelines

UNIX pipes are great, but they do have their drawbacks. Not all software can fit directly into the UNIX pipe paradigm, and UNIX pipes don't scale well to the kind of

throughput seen in modern web traffic. However, there are alternatives.

Modern “work queue” software packages have sprung up in recent years, allowing for rudimentary FIFO queues that work **across machines**. Packages like `beanstalkd` and `celery` allow for the creation of arbitrary work queues between processes. These can easily simulate the behaviour of traditional UNIX pipes, and have the major advantage of being **distributed** across many machines. However, they’re fairly well suited to *asynchronous* task processing, and their queues typically don’t block processes that try to send messages, which doesn’t allow for the kind of implicit execution control we saw earlier with UNIX pipes. These services act more as messaging systems and work queues rather than as coroutines.

To work around this lack of synchronization and pressure in distributed pipeline systems, I’ve created my own project - a Redis-based, reliable, distributed synchronized pipeline library called **pressure**. **pressure** allows you to set up pipes between different processes, but adds the ability to have pipe buffers **persist** and be used **across multiple machines**. By using Redis as a stable message broker, all of the inter-process communication is taken care of and is **OS and platform agnostic**. (Redis also gives a bunch of nice features like reliability and replication.)

pressure’s reference implementation is in Python, and it’s still in its infancy. To show its power, let’s try to replicate the pipeline example from the start of this post by using **pressure**’s included UNIX pipe adapter. (`put` and `get` are small C programs that act as a bridge between traditional UNIX pipes and distributed **pressure** queues kept in Redis.)

```
# Read in the system's dictionary
cat /usr/share/dict/words | ./put test_1 &
```

```
# Find words containing 'purple'
./get test_1 | grep purple | ./put test_2 &

# Count the letters in each word
./get test_2 | awk '{print length($1), $1}' | ./put test_3 &

# Sort lines
./get test_3 | sort -n | ./put test_4 &

# Take the last line of the input
./get test_4 | tail -n 1 | ./put test_5 &

# Take the second part of each line
./get test_5 | cut -d " " -f 2 | ./put test_6 &

# Put the resulting word into Tux's mouth
./get test_6 | cowsay -f tux
```

The first thing to note - this is an **extremely** slow operation, as we're filtering a multi-megabyte file with this method. We end up sending 235,912 messages through Redis, which takes the better part of 4 minutes. (If we move `grep` to run immediately after `cat`, and before putting data into Redis, this operation runs more than 1,200 times faster.) However, in the end, we get the correct answer that we're looking for:

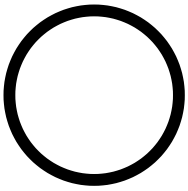
```
 _____________
< unimpurpled >
 -------------
   \
    \
        .--.
       |o_o |
       |:_/ |
      //   \ \
     (|     | )
    /'\_   _/`\
    \___)=(___/
```

However, another peculiar property can be observed - by logging in to `redis-cli`, the Redis command line tool, we can find out that very little memory is being used by our pipeline, despite the large set of input data:

```
$ redis-cli info | grep memory
used_memory:3126928
used_memory_human:2.98M
used_memory_rss:2850816
used_memory_peak:3127664
used_memory_peak_human:2.98M
used_memory_lua:31744
```

pressure is still in alpha, and definitely **not** ready for wide-scale production deployment, but you should definitely try it out!

Pipelines are hugely useful tools in software that can help cut down on resource usage. Bounded pipelines act as **coroutines** to only do computation when necessary, and can be crucial in certain applications, like real-time audio processing. **pressure** provides a way to easily use pipelines reliably on multiple machines. Try using the pipes-and-filters paradigm to solve your own software architecture problems, and see how simple and efficient it can be!



1,092 KUDOS

NOW READ THIS

The middle ground between form and function

I've noticed a distinct trend in all of my recent work. Not all of it is useful, and not all of it is feature-complete - but it all places a lot of importance on form over function. Let me give an

example: Earlier this month, I put... **Continue →**

@psobot say hello petersobot.com

SVBTLE

Terms • Privacy • Promise