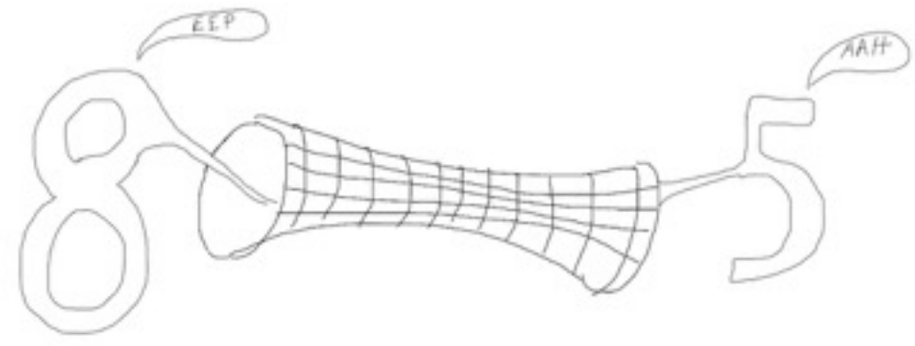
```
open("/dev/bangbangcon", O_SPEAK);
```

Hi everybody, I'm Andreas and I'm here to talk about the UNIX Input / Output system.

There is one particular aspect of the UNIX I/O system that everybody kinda agrees on: The idea that Everything Is A File.

Usually when we talk about files, we think of entries in the file system - some pathname that contains bytes. These dormant file system entries aren't the kind of file I'll be talking about today.

This talk is about the files that processes have opened, that they are reading or writing. These are called file descriptors.



I'm not a number, I'm a free file descriptor!!1

(Our protagonist promptly disappears down a wormhole)
@antifuchs, https://boinkor.net/go/fdfun

One day I was talking about UNIX with my colleague Nelson, who knows kernels.

He sent me down a path to see what weird things UNIX lets you do with I/O. UNIX does not disappoint.

I found mechanisms to circumvent limits and break down process isolation, ways to bring a machine to its knees, and some very fun implementation details in the Linux and OS X kernels.

File Descriptors

They're more than just numbers.

File descriptors are really the concept behind how UNIX does input and output.

To an application they are just numbers, but they hide a massive amount of complexity. And they also let you do incredibly weird and funny things.

Now, a number alone doesn't carry a whole lot of information. It's just a handle to an entry in a table somewhere that holds the file descriptor's state, and that's managed by the operating system.

This is a primitive but simple API design: Numbers are very easy to pass around for programs!
The OS can keep track of all that state in a table and easily retrieve it using the number as an offset.

They're not just files either

- Pipes
- Sockets
- Terminals
- In Linux, weird things: memfd, eventfd, signalfd, timerfd

While we call them file descriptors, they can refer to more than just files!

Any kind of Input or Output on UNIX systems usually goes through a file descriptor.

The operation that returns a file descriptor usually varies - for regular files it's open(2), but for sockets it's socket(2) and so on.

Let's look at an example of what files a process can have open!

```
asf@bonnetmaker:~> cat /dev/urandom | grep "testing" - >/tmp/rando-results & sudo lsof -p $!
[1] 402
COMMAND PID USER   FD   TYPE DEVICE SIZE/OFF     NODE NAME
grep    402  asf  cwd    DIR  259,2     4096  1310749 /home/asf
grep    402  asf  rtd    DIR  259,2     4096        2 /
grep    402  asf  txt    REG  259,2   215320  8915217 /bin/grep
grep    402  asf  mem    REG  259,2  2981280  7866216 /usr/lib/locale/locale-archive
grep    402  asf  mem    REG  259,2   138744  4719266 /lib/x86_64-linux-gnu/libpthread-2.23.so
grep    402  asf  mem    REG  259,2  1864888  4719267 /lib/x86_64-linux-gnu/libc-2.23.so
grep    402  asf  mem    REG  259,2    14608  4719268 /lib/x86_64-linux-gnu/libdl-2.23.so
grep    402  asf  mem    REG  259,2   456632  4723346 /lib/x86_64-linux-gnu/libpcre.so.3.13.2
grep    402  asf  mem    REG  259,2   162632  4719172 /lib/x86_64-linux-gnu/ld-2.23.so
grep    402  asf  mem    REG  259,2    26258  7873075 /usr/lib/x86_64-linux-gnu/gconv/gconv-modules.cache
grep    402  asf    0r  FIFO   0,10      0t0 16132498 pipe
grep    402  asf    1w   REG  259,2        0  2883608 /tmp/rando-results
grep    402  asf    2u   CHR  136,6      0t0        9 /dev/pts/6
asf@bonnetmaker:~>
```

lsof

This is the output of a tool called lsof. Its name is short for "list open files". It shows a bunch more stuff, but I'll focus on the open file descriptors on the last three lines.

In this example, you can see that I was running grep on a pipe pipe opened as file descriptor zero for reading, redirecting output to a file that is opened on file descriptor one.

That's all conventions - these numbers are meaningful to the C library. Everything after file descriptor number 2 belongs to the program.

Files that one process has open can also be passed to other processes.

The easiest way to pass files around between processes is inheritance - a child process will inherit a parent's open files.

Orrrrr...

UNIX Domain Sockets

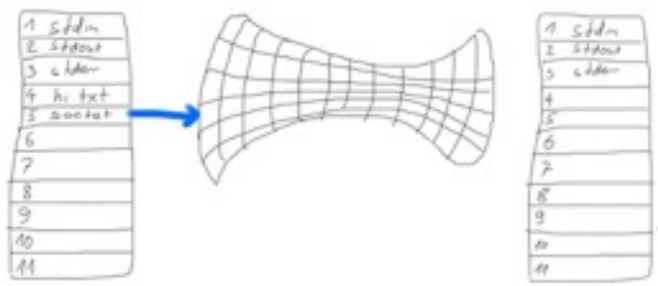
Like a wormhole, for your open files!

You can also pass files between arbitrary processes, using UNIX Domain sockets.

The API is a bit complicated, so I'll gloss over some parts.

To a programmer, they look like network sockets that you connect to via a pathname like /tmp/mysql.sock.

But they work only on the local system, and they're guaranteed to deliver every message you send, in-order.

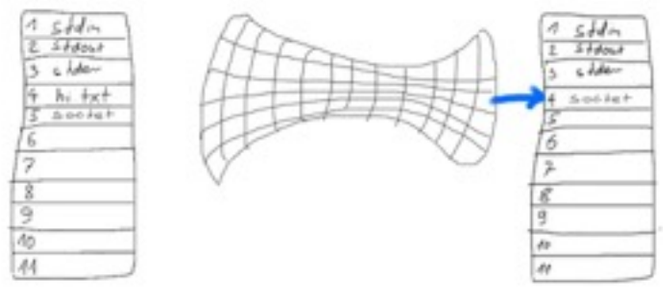


Totally a wormhole.

And they let you send files from one process to another!

A process can send a file descriptor across a UNIX Domain Socket by sending a message that has out-of-band data, containing control messages mentioning those file descriptors.

(slide)



Totally a wormhole.

(Including time travel)

The operating system does the rest and makes sure that the other end gets the open file onto its file table as soon as it receives the message.

The timeline here is interesting - the sending process could close the file before the other process receives it.

It's almost like time travel!

This is pretty cool!

- The kernel keeps files open for you?!
- UNIX sockets are identified by file descriptors too!
- Every process has a limit on open files

That your file descriptors stay open can mean only one thing: There's another table inside the kernel that they live in, and the kernel closes them when all their references are closed.

And also - yep, sockets are file descriptors. And yep, you can send them down a Unix Domain Socket. Socket recursion is totally a thing.

People who design I/O layers like the one in UNIX put buffers and limits everywhere. So there is a number of files your program can have open.

What if I told you that the limit on how many open files a process can have could be circumvented?

You totally can circumvent the open file limit.

- OS X: 7168 open files per process.
- Linux: 1024.
- What if you need more? Close them.
- What if you need them to remain open? 😅💍

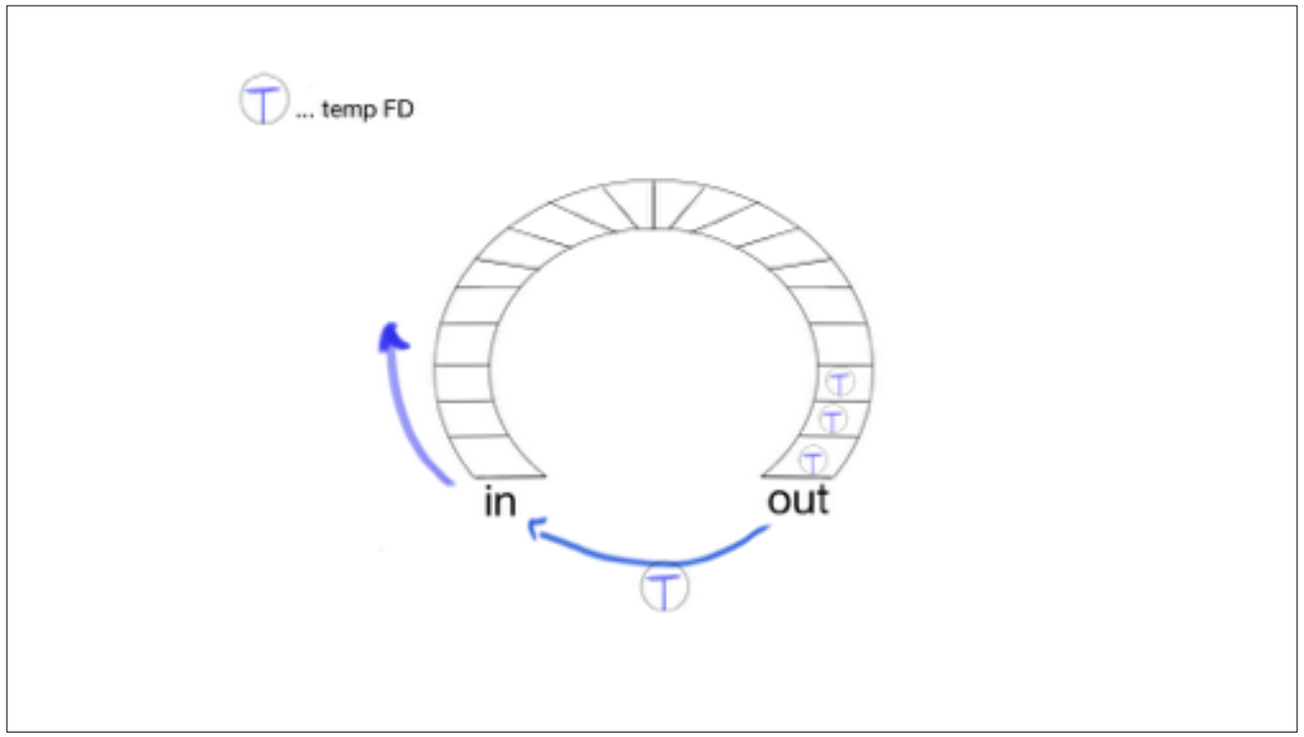
You totally can circumvent that limit.

OS X and Linux have different defaults, but the principle is the same: You get a few thousand open files and then you have to budget.

Some processes sometimes need to handle more, and sometimes the owner of the machine doesn't like to adjust that limit. What do you do then?!

There's a workaround: You could make a pair of UNIX domain sockets, both connected to your process, send open files into one and then close the files it sent!

Once it needs an open file back, it reads from the other end and puts them back into the first socket until it finds the one it needs.



That starts to look like a ring buffer, and that's what I called them:

The test program I wrote opens two UNIX sockets using `socketpair`: one to put files in, the other to get them out.

Then you can send files into one socket and close them. They come out, in order, from the other end.

This works really well across platforms!

How many more files can we have open with just one?

```
Hit EMFILE: Too many open files, aborting
I managed to store a bunch of FDs in #<Ring containing 555 fds>
...and I opened 4994 FDs
```

Linux

Limit: 5000
Actual: 5549

In Linux, we get about 500 file descriptors into that ring buffer before it's full.

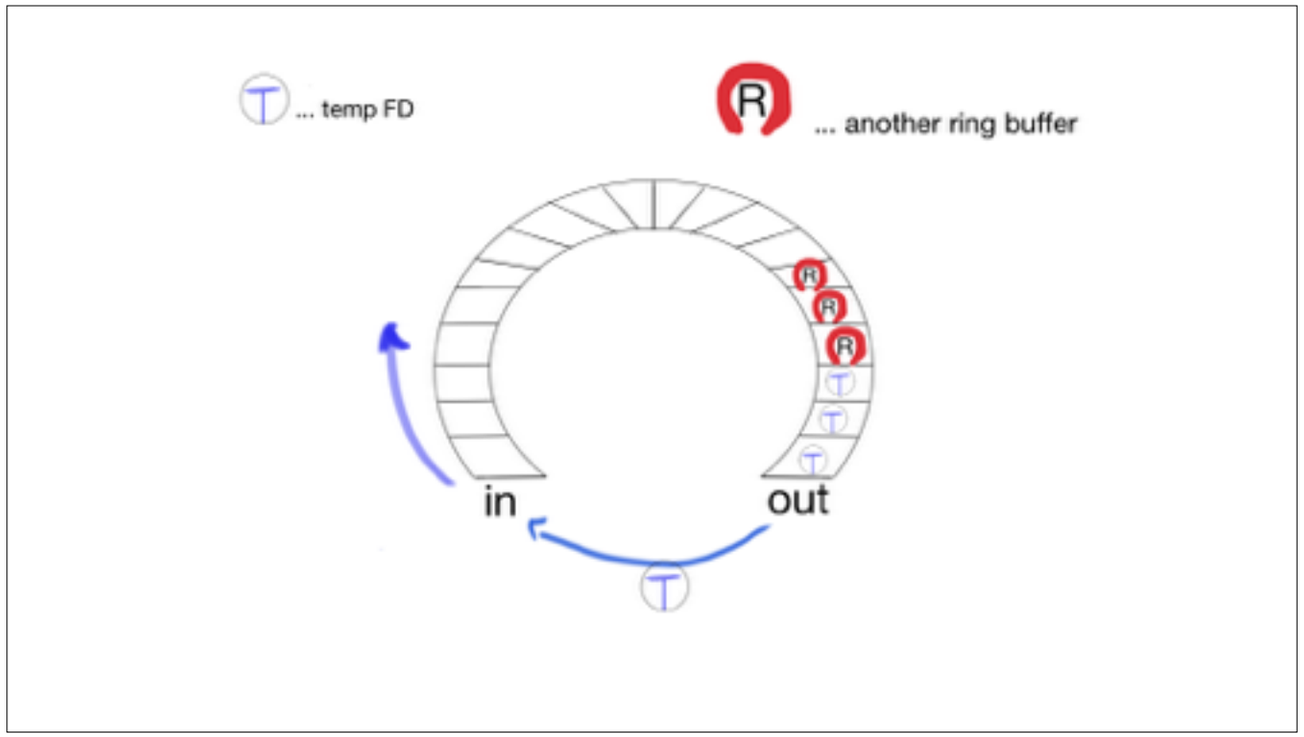
```
I hit EAGAIN: Try again
Hit EMFILE: Too many open files, aborting
I managed to store a bunch of FDs in #<Ring containing 3600 fds>
...and I opened 4992 FDs
```

OS X

Limit: 7168
Open: 8592

On OS X, we can open about 1.5k more sockets than we normally could!

But can we take this further?



Yes, of course! You can make more rings and recursively stuff them into other rings!

How many file descriptors are those, you ask?

```
I hit ENFILE: File table overflow - this means something global is full, probably
Assembled an outer ring of 4ring containing 365 entries» for a total of 202575 FDs, lsof output follows:
COMMAND   PID USER   FD   TYPE             DEVICE SIZE/OFF    NODE NAME
ring_in_r  71 root  cwd    DIR              0,226      104   58410 /src
ring_in_r  71 root  rtd    DIR              0,226      180     256 /
ring_in_r  71 root  txt    REG              0,226  2106833   58550 /src/target/debug/ring_in_ring-2ae0be5818b01cf5
ring_in_r  71 root  mem    REG                0,20            1152 /lib/x86_64-linux-gnu/ld-2.19.so (path dev=0,226)
ring_in_r  71 root  mem    REG                0,20            1188 /lib/x86_64-linux-gnu/libdl-2.19.so (path dev=0,226)
ring_in_r  71 root  mem    REG                0,20            1262 /lib/x86_64-linux-gnu/libpthread-2.19.so (path dev=0,226)
ring_in_r  71 root  mem    REG                0,20            1196 /lib/x86_64-linux-gnu/libgcc_s.so.1 (path dev=0,226)
ring_in_r  71 root  mem    REG                0,20            1172 /lib/x86_64-linux-gnu/libc-2.19.so (path dev=0,226)
ring_in_r  71 root  mem    REG                0,20            1210 /lib/x86_64-linux-gnu/libm-2.19.so (path dev=0,226)
ring_in_r  71 root  mem    REG                0,20            1268 /lib/x86_64-linux-gnu/librt-2.19.so (path dev=0,226)
ring_in_r  71 root  mem    REG                0,20           58550 /src/target/debug/ring_in_ring-2ae0be5818b01cf5 (path dev=0,226)
ring_in_r  71 root    0u   CHR              136,0      0t0       3 /dev/pts/0
ring_in_r  71 root    1u   CHR              136,0      0t0       3 /dev/pts/0
ring_in_r  71 root    2u   CHR              136,0      0t0       3 /dev/pts/0
ring_in_r  71 root    3u  unix 0xffff88007b8d4440      0t0 4167615 socket
ring_in_r  71 root    4u  unix 0xffff88007b8d4880      0t0 4167616 socket
ring_in_r  71 root    7r  FIFO               0,10      0t0 4171724 pipe
ring_in_r  71 root    9r  FIFO               0,10      0t0 4171725 pipe
```

Linux: 😎

202,575 open files?!

In Linux, this technique allows us to open 200k files before the OS couldn't open more files.

This is a pretty great way to run a whole operating system into the ground - your process's misbehavior can really confuse or crash other processes on the system.

The additional files don't even appear in the list of the process's file table, so it'll even confuse systems people trying to debug this!

So Linux tried to be very helpful to my test program, but on OS X, I hit some unexpected behavior!

OS X does not like this

- The program can receive the "inner" ring from the "outer" ring
- It can receive the messages I sent into the "inner" ring
- None of the messages received has open files on it.

Everything was looking great on OS X: I could send files into a ring buffer and close them. I could send that ring buffer into another ring buffer and close the inner ring's socket pair.

Then, when I received the inner buffer, it had messages on it that I'd sent, but no file descriptors came with those messages.

Whaaaaat.

Conclusion: OS X does reference counting on sockets

The only conclusion I can draw (and then confirmed by a chat with my colleague Nelson, who knows kernels) is that BSD-alike kernels like OS X close all sockets "trapped" in a UNIX socket pair when that socket pair gets closed.

Any time I closed rings one layer deep, the contained files were closed.

It does not matter that the ring is itself still trapped in another UNIX socket ring.

That's kinda reasonable.

This program's behavior is pretty far out, and closing files is better than leaking them!

But wait: Linux does socket Garbage Collection?!

But... if OS X closes trapped file descriptors, why does this work on Linux?

It's because apparently (and again, according to Nelson), Linux has a garbage collector for sockets, traversing file descriptors contained in open sockets recursively.

I hear it does Mark & Sweep.

I find this amazing. The Linux kernel has a garbage collector that ensures that my ridiculous test program can do its thing. Thank you, Linux! That's very considerate!

Lesson learned:

Weird 🖥 behavior is awesome.

By building these tiny test programs and trying out how they break, we could take a mostly opaque thing and figure out how it works!

We even found some possibly unintended consequences and pretty pathological behavior in the process!

Finding this kind of thing always makes me, personally, very happy.

Thank you!

Get my code: https://boinkor.net/go/fdfun
Tell me about it: @antifuchs

Thanks for listening to me - I hope you enjoyed my talk!

If you want to try out my test code, you can find it on github. It's in rust and uses libraries that Kamal, our next speaker has worked on!

The short link boinkor.net/go/fdfun takes you to there - you can find my slides there, too.

If you do try it, I would love to hear from you! I'm @antifuchs on twitter, or if you're here, come talk to me!