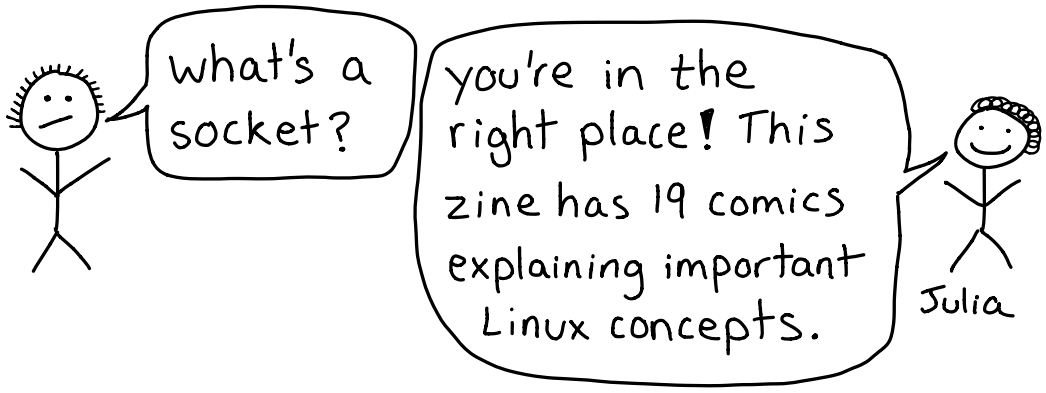
love this?
find more awesome zines at
→ wizardzines.com ←



... 5 minutes later ...

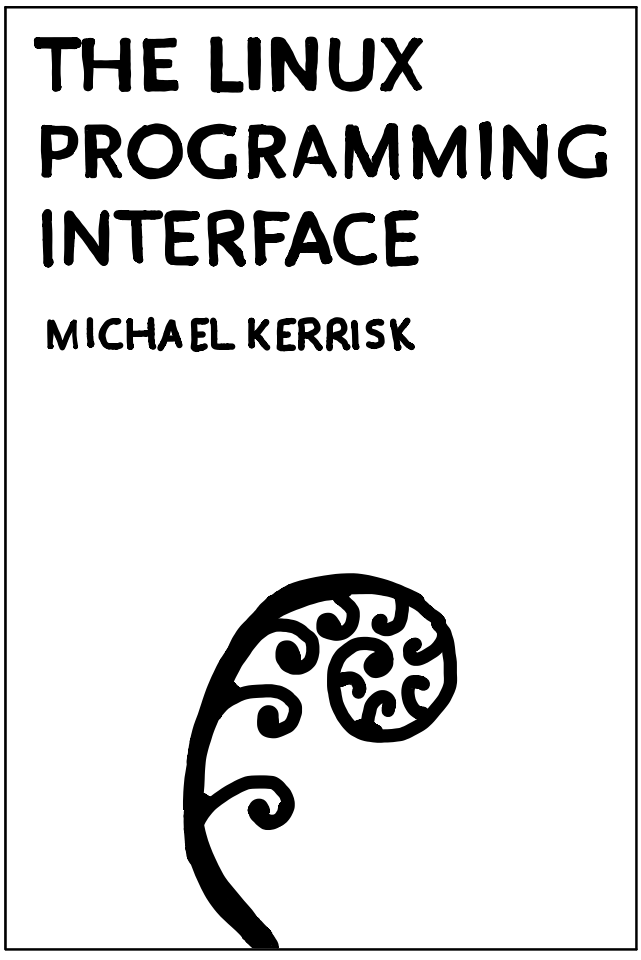
by Julia Evans
https://jvns.ca
twitter.com/b@rk

Want to learn more? I highly recommend this book: ⟶

Every chapter is a readable, short (usually 10-20 pages) explanation of a Linux system.

I used it as a reference constantly when writing this zine.

I ♡ it because even though it's huge and comprehensive (1500 pages!), the chapters are short and self-contained and it's very easy to pick it up and learn something.



man page sections

man pages are split up into 8 sections

① ② ③ ④ ⑤ ⑥ ⑦ ⑧

$ man 2 read

means "get me the man page for read from section 2".

There's both
→ a program called "read"
→ and a system call called "read"

so

$ man 1 read

gives you a different man page from

$ man 2 read

If you don't specify a section, man will look through all the sections & show the first one it finds.

man page sections

① programs
$ man grep
$ man ls

② system calls
$ man sendfile
$ man ptrace

③ C functions
$ man printf
$ man fopen

④ devices
$ man null
for /dev/null docs

⑤ file formats
$ man sudoers
for /etc/sudoers
$ man proc
files in /proc!

⑥ games
not super useful.
$ man sl
is my favourite from that section

⑦ miscellaneous
explains concepts!
$ man 7 pipe
$ man 7 symlink

⑧ sysadmin programs
$ man apt
$ man chroot

♥ Table of contents ♥

unix permissions

There are 3 things you can do to a file

read write execute

ls -l file.txt shows you permissions. Here's how to interpret the output:

rw- rw- r-- bork staff

- rw-: bork (user) can read & write
- rw-: staff (group) can read & write
- r--: ANYONE can read

File permissions are 12 bits

setuid setgid

	user	group	all
000	110	110	100
sticky	rwx	rwx	rwx

For files:
- r = can read
- w = can write
- x = can execute

For directories, it's approximately:
- r = can list files
- w = can create files
- x = can cd into & access files

110 in binary is 6

So rw- r-- r--
= 110 100 100
= 6 4 4

chmod 644 file.txt means change the permissions to:

rw- r-- r--

Simple!

setuid affects executables

$ls -l /bin/ping

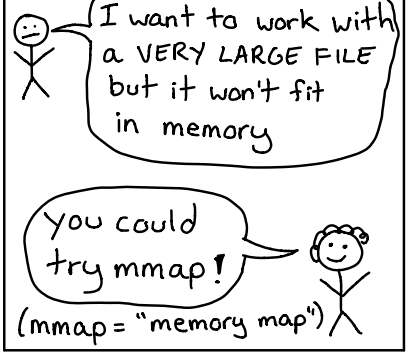
rws r-x r-x root root

this means ping always runs as root

setgid does 3 different unrelated things for executables, directories, and regular files.

mmap

what's mmap for?



load files lazily with mmap

When you mmap a file, it gets mapped into your program's memory.

but nothing is ACTUALLY read into RAM until you try to access the memory. (how it works: page faults!)

how to mmap in Python

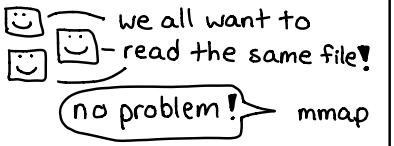
```
import mmap
f = open("HUGE.txt")
mm = mmap.mmap(f.fileno(), 0)
```

this won't read the file from disk! Finishes ~instantly.

```
print(mm[-1000:])
```

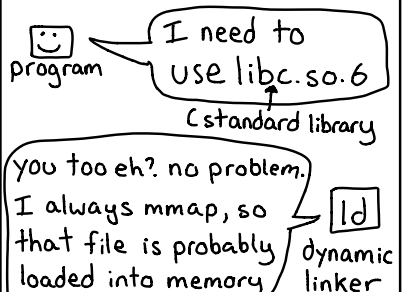
this will read only the last 1000 bytes!

sharing big files with mmap



Even if 10 processes mmap a file, it will only be read into memory ♥once♥

dynamic linking uses mmap



anonymous memory maps

- not from a file (memory set to 0 by default)
- with MAP_SHARED, you can use them to share memory with a subprocess!

page faults

every Linux process has a page table

★ page table ★

virtual memory address	physical memory address
0x19723000	0x1422000
0x19724000	0x1423000
0x1524000	not in memory
0x1844000	0x4a000 read only

some pages are marked as either

★ read only

★ not resident in memory

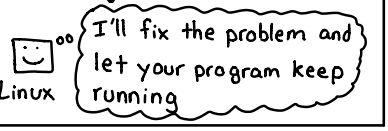
when you try to access a page that's marked "not resident in memory", it triggers a ! page fault !

What happens during a page fault?

→ the MMU sends an interrupt

→ your program stops running

→ Linux kernel code to handle the page fault runs

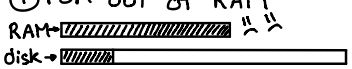


"not resident in memory" usually means the data is on disk !

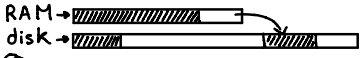
Having some virtual memory that is actually on disk is how swap and mmap work.

how swap works

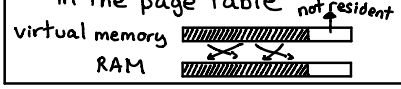
① run out of RAM



② Linux saves some RAM data to disk

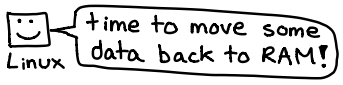


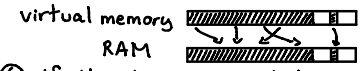
③ mark those pages as "not resident in memory" in the page table



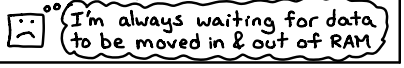
④ When a program tries to access the memory, there's a ! page fault !

⑤





⑥ if this happens a lot, your program gets VERY SLOW



an amazing directory: /proc

Every process on Linux has a PID (process ID) like 42.

In /proc/42, there's a lot of VERY USEFUL information about process 42.

/proc/PID/cmdline

command line arguments the process was started with

/proc/PID/exe

symlink to the process's binary. magic: works even if the binary has been deleted !

/proc/PID/environ

all of the process's environment variables

/proc/PID/status

Is the program running or asleep? How much memory is it using? And much more!

/proc/PID/fd

Directory with every file the process has open!

Run `$ ls -l /proc/42/fd` to see the list of files for process 42.

These symlinks are also magic & you can use them to recover deleted files ♥

/proc/PID/stack

The kernel's current stack for the process. Useful if it's stuck in a system call.

/proc/PID/maps

List of process's memory maps. Shared libraries, heap, anonymous maps, etc.

and more

Look at

man proc

for more information !

system calls

The Linux kernel has code to do a lot of things

read from a hard drive

make network connections

create new process

kill process

change file permissions

keyboard drivers

your program doesn't know how to do those things

TCP? dude I have no idea how that works.

NO, I do not know how the ext4 filesystem is implemented. I just want to read some files!

programs ask Linux to do work for them using =system calls=

program: please write to this file

<switch to running kernel code>

Linux: done! I wrote 1097 bytes!

<program resumes>

every program uses system calls

Python program: I use the 'open' syscall to open files

Java program: me too!

C program: me three!

and every system call has a number (e.g. chmod is #90 on x86-64)

So what's actually going on when you change a file's permissions is:

program: run syscall #90 with these arguments

Linux: ok!

you can see which system calls a program is using with {strace}

```
$ strace ls /tmp
```

will show you every system call 'ls' uses! it's really fun!

⚠ strace has high overhead so don't run it on your production database

copy on write

On Linux, you start new processes using the fork() or clone() system call.

calling fork creates a child process that's a copy of the caller

parent

child

the cloned process has EXACTLY the same memory.

- → same heap
- → same stack
- → same memory maps

if the parent has 3GB of memory, the child will too.

copying all that memory every time we fork would be slow and a waste of RAM

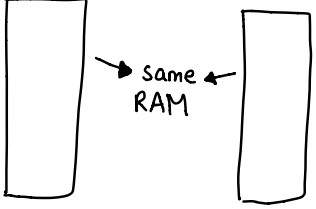
often processes call exec right after fork, which means they don't use the parent process's memory basically at all!

so Linux lets them share physical RAM and only copies the memory when one of them tries to write

process: I'd like to change that memory

Linux: okay! I'll make you your own copy!

Linux does this by giving both the processes identical page tables.



but it marks every page as read only.

when a process tries to write to a shared memory address:

① there's a =page fault=

② Linux makes a copy of the page & updates the page table

③ the process continues, blissfully ignorant

process: It's just like I have my own copy

shared libraries

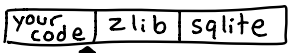
Most programs on Linux use a bunch of C libraries.

some popular libraries:

- openssl (for SSL!)
- sqlite (embedded db!)
- libpcre (regular expressions!)
- zlib (gzip!)
- libstdc++ (C++ standard library!)

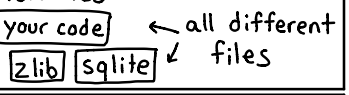
There are 2 ways to use any library:

1. Link it into your binary

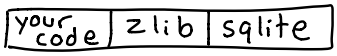


big binary with lots of things!

2. Use separate shared libraries

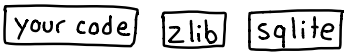


Programs like this

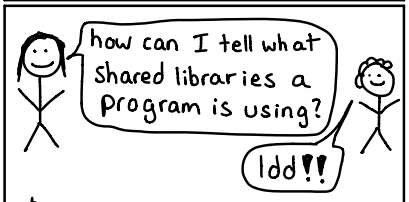


are called "statically linked"

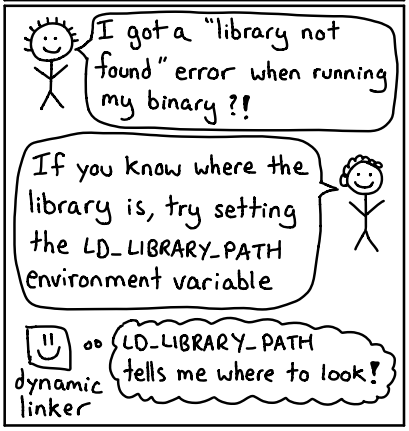
and programs like this



are called "dynamically linked"




```
$ ldd /usr/bin/curl
    libz.so.1 => /lib/x86-64...
    libresolv.so.2 => ...
    libc.so.6 => ...
    +34 more
```

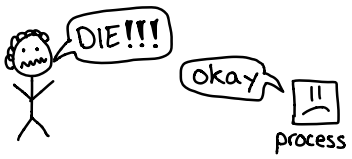


Where the dynamic linker looks

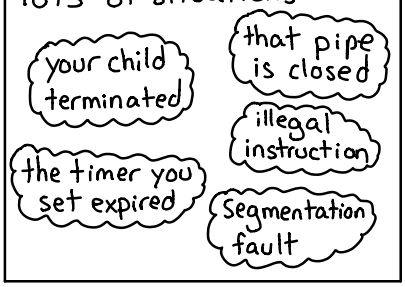
1. DT_RPATH in your executable
2. LD_LIBRARY_PATH
3. DT_RUNPATH in executable
4. /etc/ld.so.cache (run ldconfig -p to see contents)
5. /lib, /usr/lib

signals

If you've ever used kill you've used signals



the Linux kernel sends processes signals in lots of situations



you can send signals yourself with the kill system call or command

SIGINT Ctrl-C
SIGTERM kill
SIGKILL kill -9
} various levels of "die"

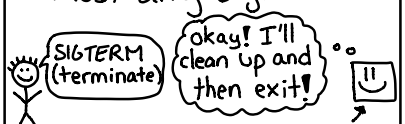
SIGHUP kill -HUP

often interpreted as "reload config", e.g. by nginx

Every signal has a default action, which is one of:

- ignore
- Kill process
- Kill process AND make core dump file
- Stop process
- resume process

Your program can set custom handlers for almost any signal

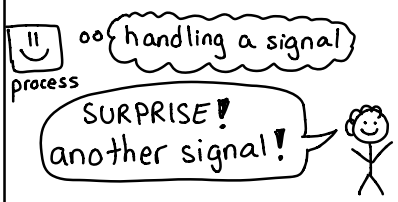


exceptions:

SIGSTOP & SIGKILL can't be ignored

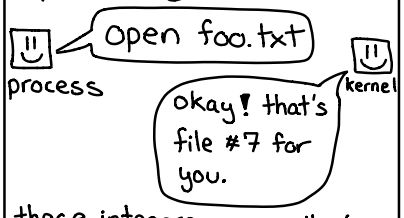
got SIGKILLed

signals can be hard to handle correctly since they can happen at ANY time



file descriptors

Unix systems use integers to track open files



these integers are called file descriptors

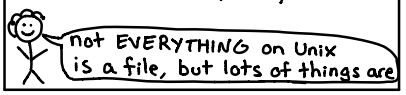
lsof (list open files) will show you a process's open files

```
$ lsof -p 4242   <- PID we're interested in
FD   NAME
 0   /dev/pts/tty1
 1   /dev/pts/tty1
 2   pipe:29174
 3   /home/bork/awesome.txt
 5   /tmp/
```

FD is for file descriptor

file descriptors can refer to:

- files on disk
- pipes
- sockets (network connections)
- terminals (like xterm)
- devices (your speaker! /dev/null!)
- LOTS MORE (eventfd, inotify, signalfd, epoll, etc.)



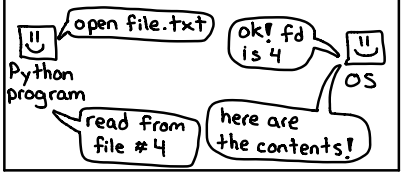
When you read or write to a file/pipe/network connection you do that using a file descriptor

Let's see how some simple Python code works under the hood:

Python:

```
f = open("file.txt")
f.readlines()
```

Behind the scenes:



(almost) every process has 3 standard FDs:

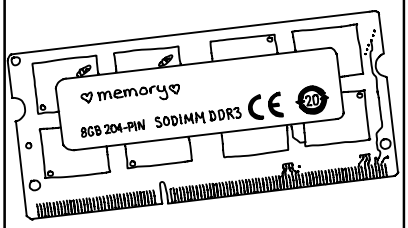
stdin → 0
stdout → 1
stderr → 2

"read from stdin" means "read from the file descriptor 0"

could be a pipe or file or terminal

virtual memory

your computer has physical memory



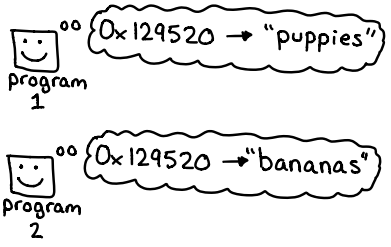
physical memory has addresses, like

0 - 8GB

but when your program references an address like 0x5c69a2a2,

that's not a physical memory address! It's a virtual address.

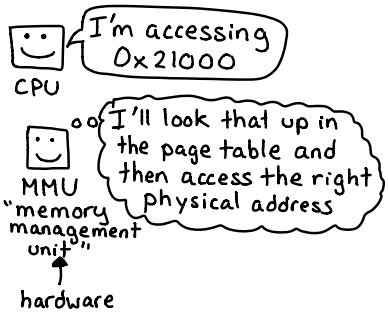
every program has its own virtual address space



Linux keeps a mapping from virtual memory pages to physical memory pages called the page table

PID	virtual addr	physical addr
1971	0x20000	0x192000
2310	0x20000	0x228000
2310	0x21000	0x9788000

when your program accesses a virtual address



every time you switch which process is running, Linux needs to switch the page table

memory allocation

your program has memory

10MB program binary

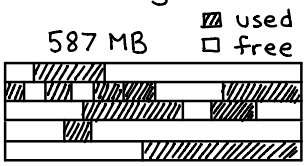
3MB stack

587MB heap

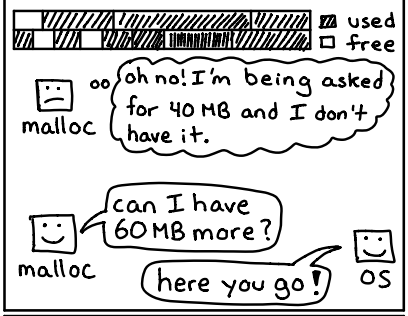
the heap is what your allocator manages

your memory allocator (malloc) is responsible for 2 things.

THING 1: keep track of what memory is used/free.



THING 2: Ask the OS for more memory!



your memory allocator's interface

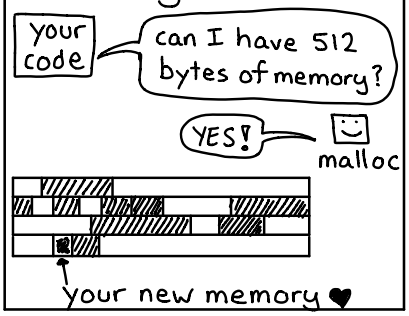
malloc(size_t size)
allocate size bytes of memory & return a pointer to it.

free(void* pointer)
mark the memory as unused (and maybe give back to the OS).

realloc(void* pointer, size_t size)
ask for more/less memory for pointer.

calloc(size_t members, size_t size)
allocate array + initialize to 0.

malloc tries to fill in unused space when you ask for memory



malloc isn't magic! it's just a function!

you can always:

→ use a different malloc library like jemalloc or tcmalloc (easy!)

→ implement your own malloc (harder)

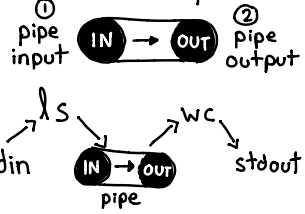
pipes

Sometimes you want to send the output of one process to the input of another

```
$ ls | wc -l
      53
```

53 files!

a pipe is a pair of 2 magical file descriptors



When ls does write(IN, "hi"), wc can read it! read(OUT) → "hi"

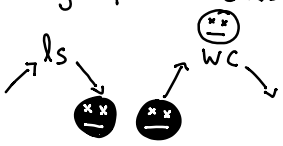
Pipes are one way. → You can't write to OUT.

Linux creates a buffer for each pipe.

If data gets written to the pipe faster than it's read, the buffer will fill up. IN [▭▭▭▭] OUT

When the buffer is full, writes to IN will block (wait) until the reader reads. This is normal & ok

what if your target process dies?



If wc dies, the pipe will close and ls will be sent SIGPIPE. By default, SIGPIPE terminates your process.

named pipes

```
$ mkfifo my-pipe
```

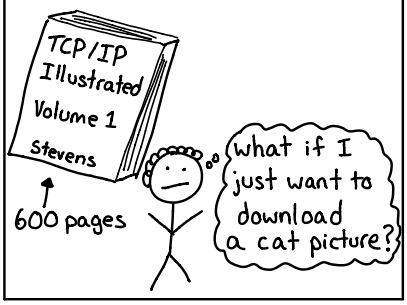
This lets 2 unrelated processes communicate through a pipe!

```
f=open(./my-pipe)
f.write("hi!\n")
```

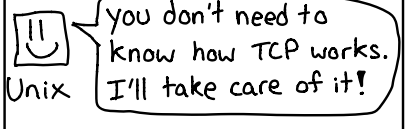
```
f=open(./my-pipe)
f.readline() ← "hi!"
```

sockets

networking protocols are complicated



Unix systems have an API called the "socket API" that makes it easier to make network connections



here's what getting a cat picture with the socket API looks like:

① Create a socket

```
fd = socket(AF_INET, SOCK_STREAM ...)
```

② Connect to an IP/port

```
connect(fd, 12.13.14.15:80)
```

③ Make a request

```
write(fd, "GET /cat.png HTTP/1.1 ...)
```

④ Read the response

```
cat_picture = read(fd ...)
```

Every HTTP library uses sockets under the hood

```
$ curl awesome.com
Python: requests.get("yay.us")
```

SOCKETS

* SO MANY edge cases though! ☹

AF_INET? What's that?

AF_INET means basically "internet socket": it lets you connect to other computers on the internet using their IP address.

The main alternative is AF_UNIX ("unix domain socket") for connecting to programs on the same computer.

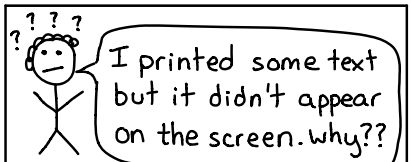
3 kinds of internet (AF_INET) sockets:

SOCK_STREAM = TCP
(curl uses this)

SOCK_DGRAM = UDP
(dig (DNS) uses this)

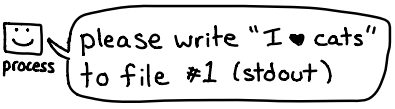
SOCK_RAW = just let me send IP packets. I will implement my own protocol.
(ping uses this)

file buffering



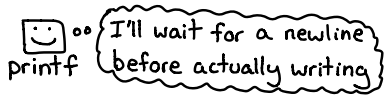
On Linux, you write to files & terminals with the system call

♥ write ♥



I/O libraries don't always call write when you print.

```
printf("I ♥ cats");
```



This is called buffering and it helps save on syscalls.

3 kinds of buffering (defaults vary by library)

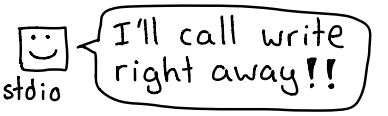
① None. This is the default for stderr.

② Line buffering. (write after newline). The default for terminals.

③ "full" buffering. (write in big c unks) The default for files and pipes.

flushing

To force your I/O library to write everything it has in its buffer right now, call flush!

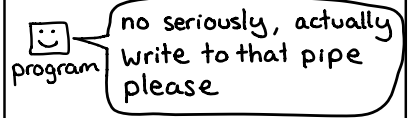


when it's useful to flush

→ when writing an interactive prompt!

```
Python example:
print("password: ", flush=True)
```

→ when you're writing to a pipe / socket



floating point

a double is 64 bits

sign exponent fraction

1001101↓ 1001101↓ 1001101↓ 1001101
1001101 1001101 1001101 1001101

$\pm 2^{E-1023} \times 1.\text{frac}$

That means there are 2^{64} doubles.

The biggest one is about 2^{1023}

weird double arithmetic

$2^{52} + 0.2 = 2^{52}$ ← (the next number after 2^{52} is $2^{52}+1$)

$1 + \frac{1}{2^{54}} = 1$ ← (the next number after 1 is $1 + \frac{1}{2^{52}}$)

$2^{2000} = \text{infinity}$ ← infinity is a double

infinity - infinity = nan ← nan = "not a number"

doubles get farther apart as they get bigger

between 2^n and 2^{n+1} there are always 2^{52} doubles, evenly spaced.

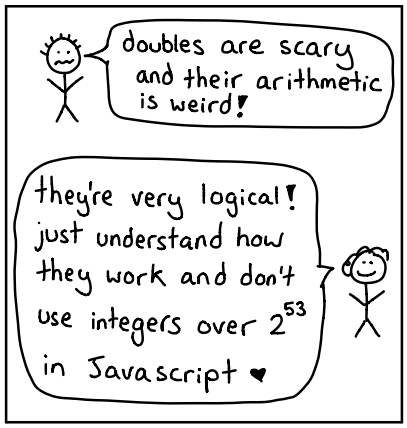
that means the next double after 2^{60} is $2^{60}+64$, $\frac{2^{60}}{2^{52}}$

Javascript only has doubles (no integers!)

```
> 2**53
9007199254740992

> 2**53+1
9007199254740992
```

same number! uh oh!



unix domain sockets

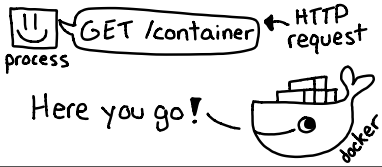
unix domain sockets are files.

```
$ file mysock.sock
socket
```

the file's permissions determine who can send data to the socket.

they let 2 programs on the same computer communicate.

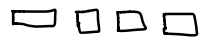
Docker uses Unix domain sockets, for example!



There are 2 kinds of unix domain sockets:

stream ← like TCP! Lets you send a continuous stream of bytes.

datagram ← like UDP! Lets you send discrete chunks of data



advantage 1

Lets you use file permissions to restrict access to HTTP/database services!

chmod 600 secret.sock

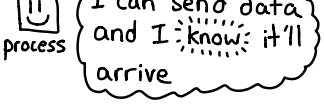
This is why Docker uses a unix domain socket.

advantage 2

UDP sockets aren't always reliable (even on the same computer).

unix domain datagram sockets are reliable!

And they won't reorder packets!



advantage 3

You can send a file descriptor over a unix domain socket.

Useful when handling untrusted input files!

What's in a process?

PID

process #129 reporting for duty!

USER and GROUP

who are you running as?

julia!

ENVIRONMENT VARIABLES

like PATH! you can set them with:

$ env A=val ./program

SIGNAL HANDLERS

I ignore SIGTERM!

I shut down safely!

WORKING DIRECTORY

Relative paths (./blah) are relative to the working directory! chdir changes it.

PARENT PID

PID 1 (init) is everyone's ancestor

PID 147

PID 129

COMMAND LINE ARGUMENTS

See them in

/proc/PID/cmdline

OPEN FILES

Every open file has an offset.

I've read 8000 bytes of that one

MEMORY

heap! stack!

shared libraries!

the program's binary!

mmaped files!

THREADS

sometimes one

sometimes LOTS

CAPABILITIES

I have CAP_PTRACE

Well I have CAP_SYS_ADMIN

NAMESPACES

I'm in the host network namespace

I have my own namespace!

container process

threads

Threads let a process do many different things at the same time

process:

thread 1: I'm calculating ten million digits of π! so fun!

thread 2: I'm finding a REALLY BIG prime number!

threads in the same process share memory

thread 1: I'll write some digits of π to 0x129420 in memory

thread 2: uh oh! that's where I was putting my prime numbers.

and they share code

calculate_pi

find_big_prime_number

but each thread has its own stack and they can be run by different CPUs at the same time

CPU 1 — π thread

CPU 2 — primes thread

sharing memory can cause problems (race conditions!)

thread 1: I'm going to add 1 to that number!

memory: 23

same time

thread 2: I'm going to add 1 to that number!

RESULT: 24 ← WRONG. Should be 25!

why use threads instead of starting a new process?

→ a thread takes less time to create.

→ sharing data between threads is very easy. But it's also easier to make mistakes with threads.

thread 1: you weren't supposed to CHANGE that data!