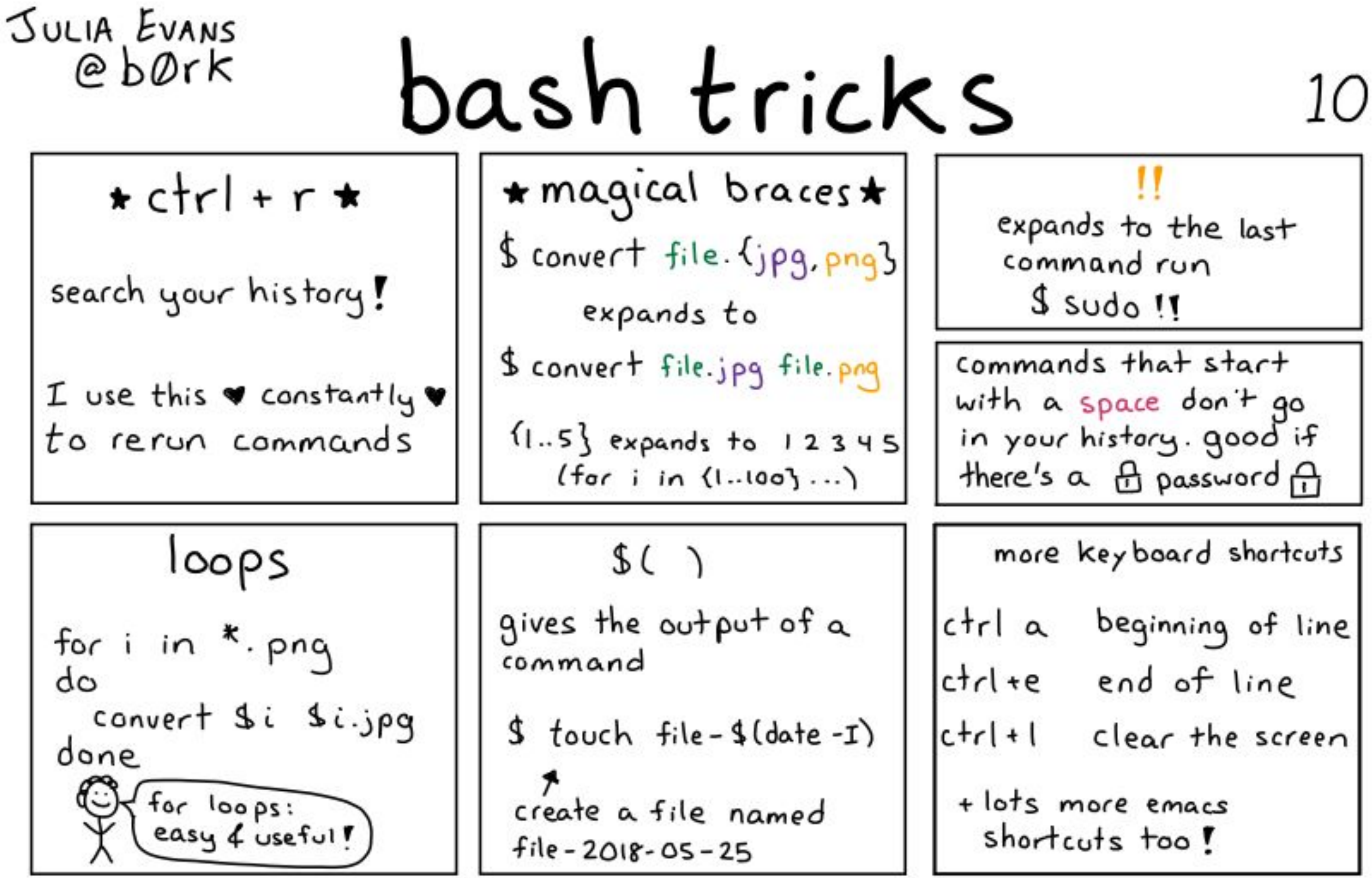
Sistemas Operativos 72.11

Entrando en calor

ITBA

Instituto Tecnológico
de Buenos Aires

Entorno de desarrollo



JULIA EVANS
@b0rk

bash tricks

10

★ ctrl + r ★

search your history!

I use this ♥ constantly ♥ to rerun commands

★ magical braces ★

```
$ convert file.{jpg,png}
```

expands to

```
$ convert file.jpg file.png
```

{1..5} expands to 1 2 3 4 5
(for i in {1..100} ...)

!!

expands to the last command run

```
$ sudo !!
```

commands that start with a space don't go in your history. good if there's a 🔒 password 🔒

loops

```
for i in *.png
do
  convert $i $i.jpg
done
```

for loops: easy & useful!

$( )

gives the output of a command

```
$ touch file-$(date -I)
```

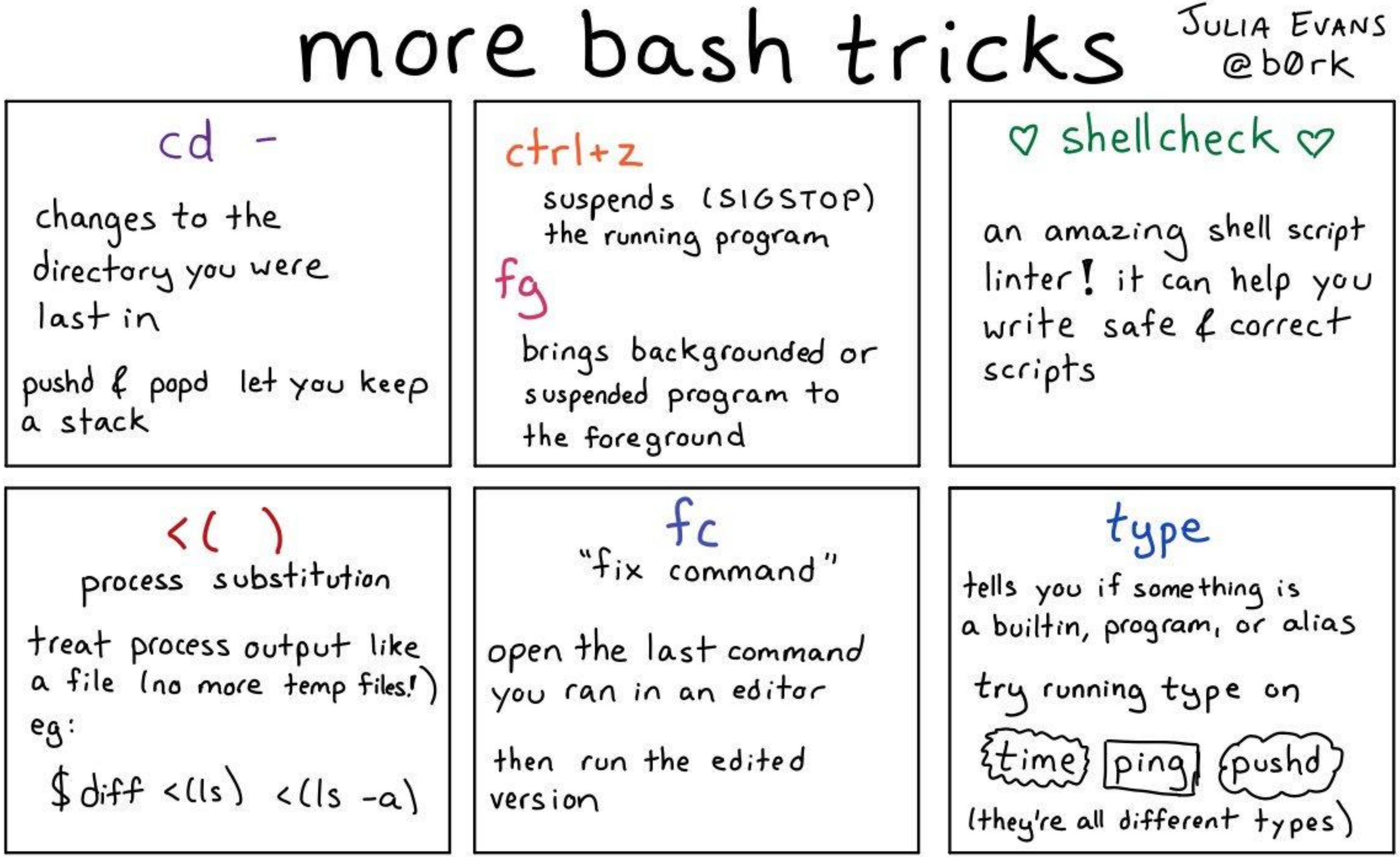
create a file named file-2018-05-25

more keyboard shortcuts

ctrl a	beginning of line
ctrl+e	end of line
ctrl+l	clear the screen

+ lots more emacs shortcuts too!

Entorno de desarrollo



more bash tricks

JULIA EVANS
@b0rk

cd -

changes to the directory you were last in

pushd & popd let you keep a stack

ctrl+z

suspends (SIGSTOP) the running program

fg

brings backgrounded or suspended program to the foreground

♡ shellcheck ♡

an amazing shell script linter! it can help you write safe & correct scripts

<( )

process substitution

treat process output like a file (no more temp files!)

eg:

```
$ diff <(ls) <(ls -a)
```

fc

"fix command"

open the last command you ran in an editor

then run the edited version

type

tells you if something is a builtin, program, or alias

try running type on

time ping pushd

(they're all different types)

Entorno de desarrollo

Docker

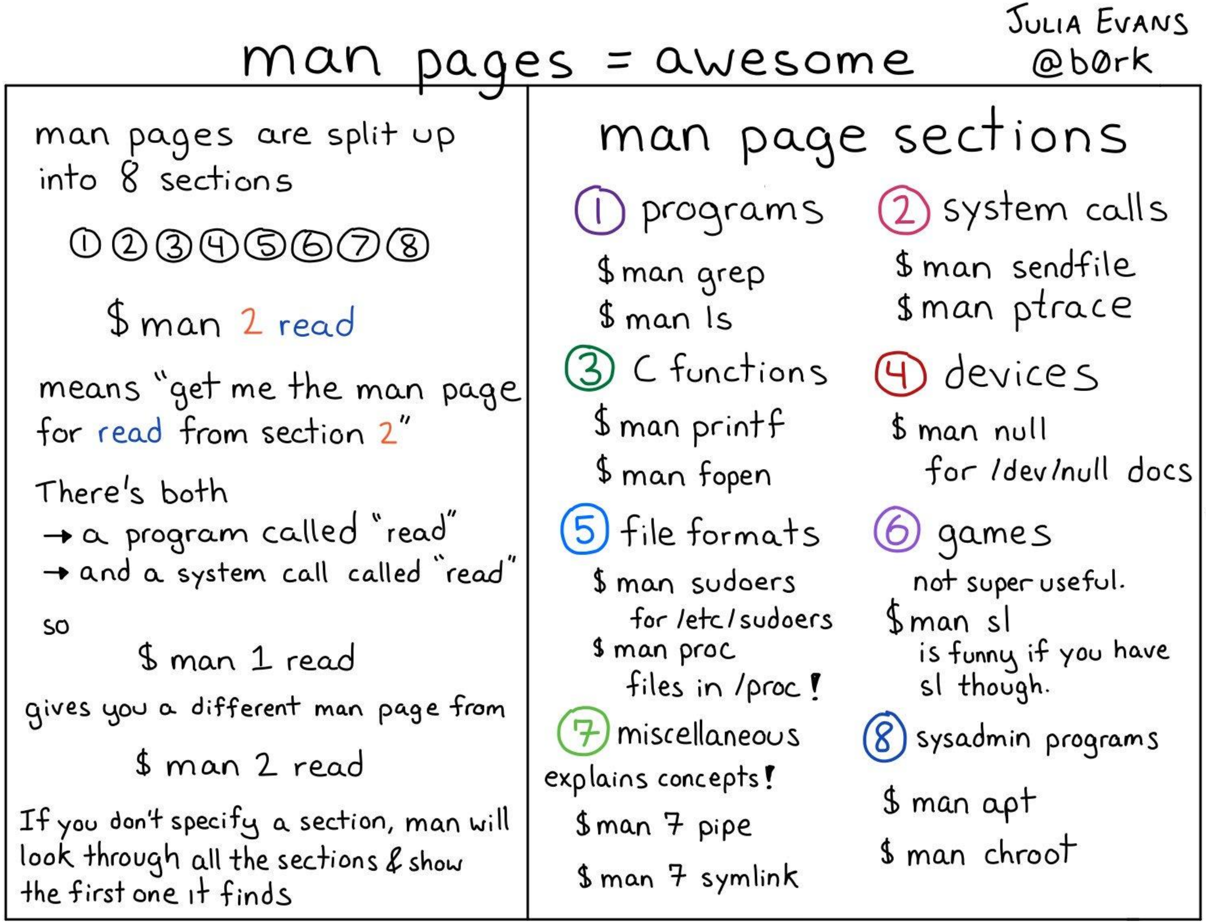
- ¿Por qué?
- Se solicitan dadores de conocimiento, factor “Docker en Windows” positivo
- Demo https://github.com/alejoaquili/ITBA-72.11-SO.git

Manuales

man

- ¿Por qué?
- Demo https://github.com/alejoaquili/ITBA-72.11-SO.git

Manuales



man pages = awesome

Julia Evans
@b0rk

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 funny if you have sl though.

⑦ miscellaneous

explains concepts!

```
$ man 7 pipe
$ man 7 symlink
```

⑧ sysadmin programs

```
$ man apt
$ man chroot
```

POSIX

fork execve wait exit

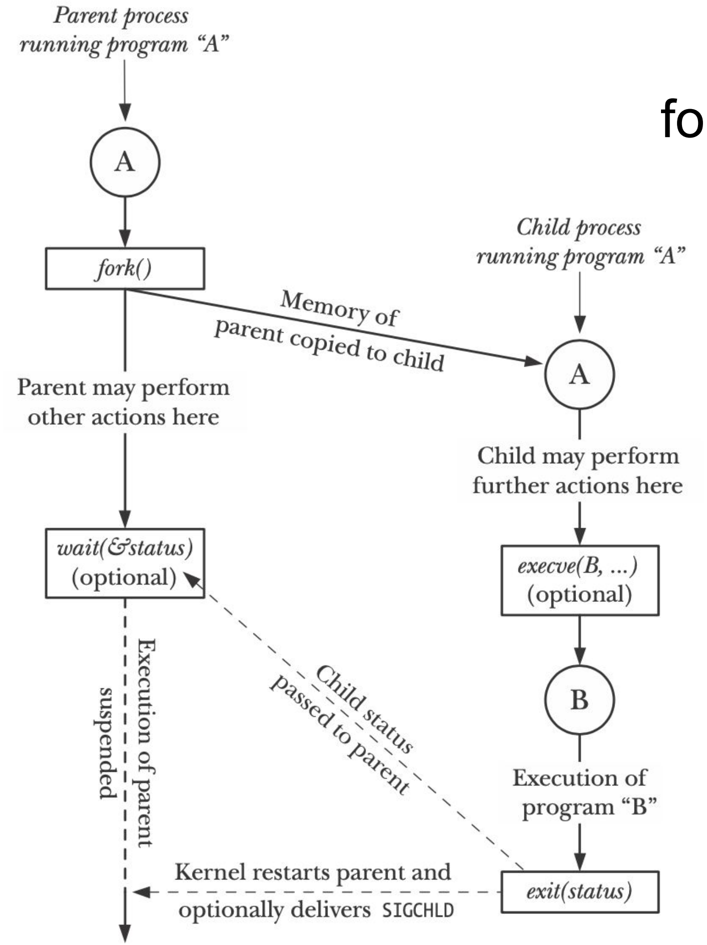

Parent process running program "A"
A
fork()
Memory of parent copied to child
Child process running program "A"
A
Parent may perform other actions here
Child may perform further actions here
wait(&status) (optional)
execve(B, ...) (optional)
B
Execution of parent suspended
Child status passed to parent
Execution of program "B"
Kernel restarts parent and optionally delivers SIGCHLD
exit(status)

Figure 24-1: Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

The Linux programming interface - Kerrisk, Michael

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

POSIX

fork execve wait exit

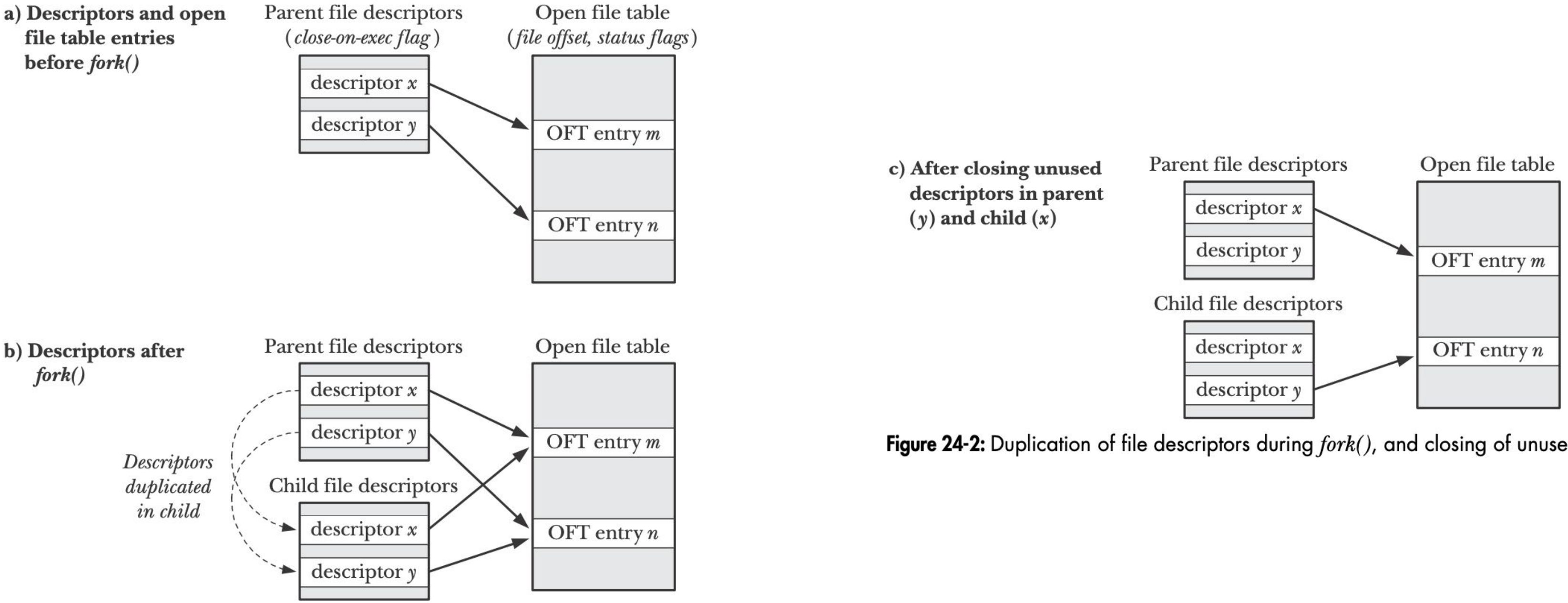


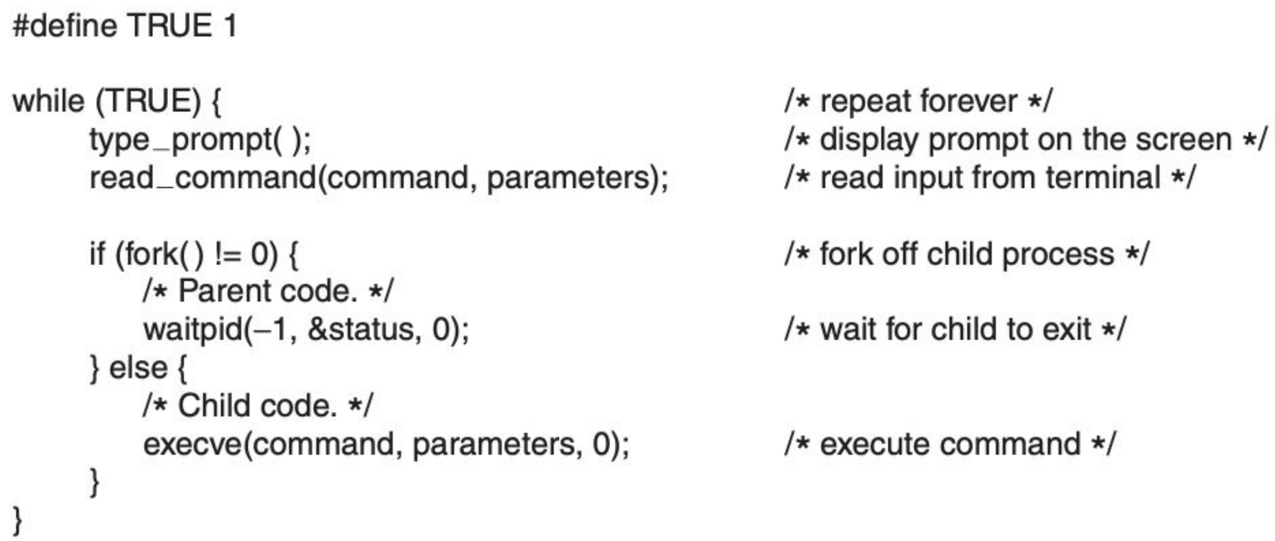
Figure 24-2: Duplication of file descriptors during *fork()*, and closing of unused descriptors

The Linux programming interface - Kerrisk, Michael

POSIX

Pair programming

- Programar una shell funcional a partir de este esqueleto

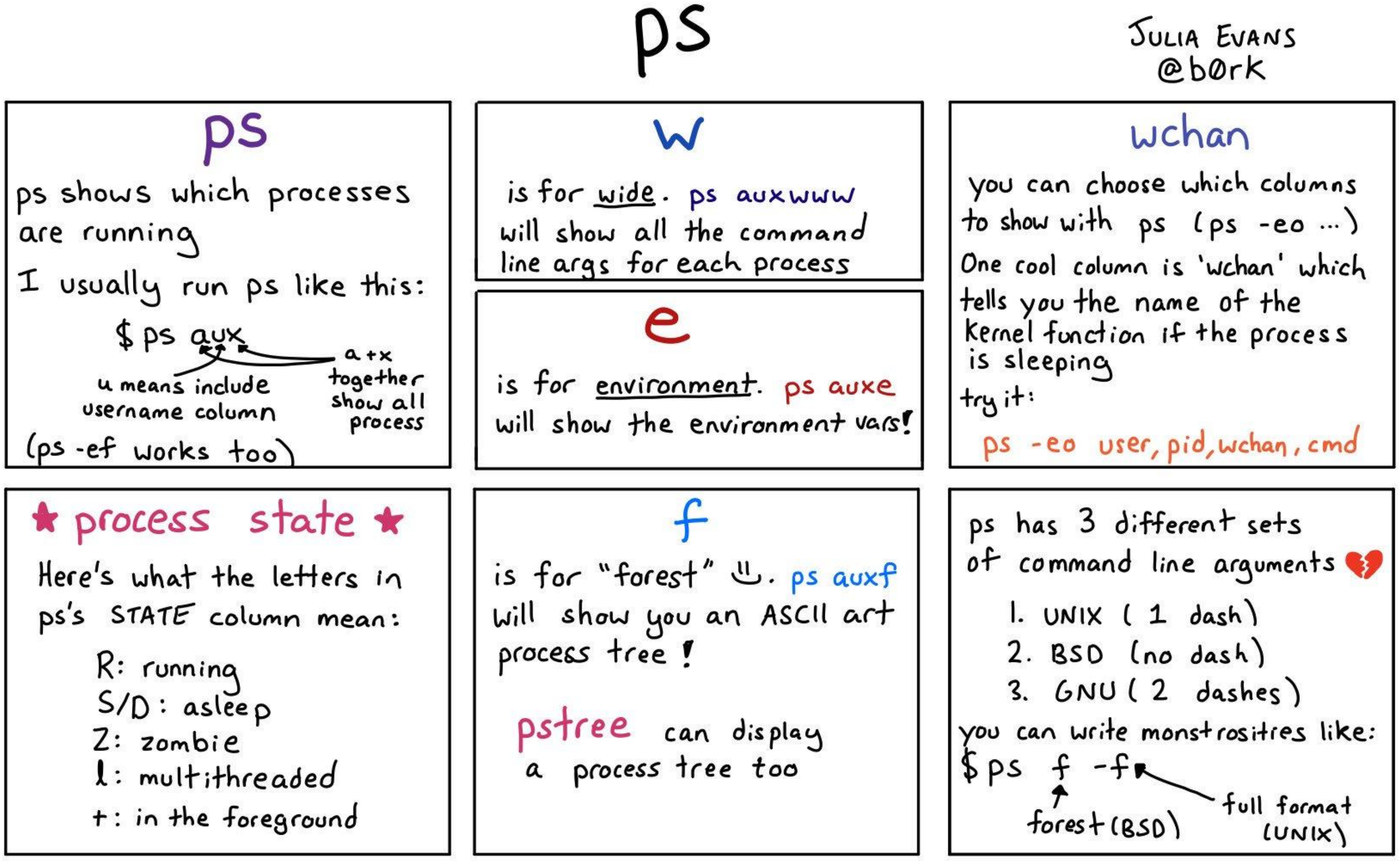


```
#define TRUE 1

while (TRUE) {                                       /* repeat forever */
      type_prompt( );                                /* display prompt on the screen */
      read_command(command, parameters);             /* read input from terminal */

      if (fork( ) != 0) {                            /* fork off child process */
          /* Parent code. */
          waitpid(–1, &status, 0);                   /* wait for child to exit */
      } else {
          /* Child code. */
          execve(command, parameters, 0);            /* execute command */
      }
}
```

Comandos UNIX



ps

JULIA EVANS
@b0rk

ps

ps shows which processes are running

I usually run ps like this:

```
$ ps aux
```

u means include username column

a+x together show all process

(ps -ef works too)

w

is for wide. ps auxwww will show all the command line args for each process

e

is for environment. ps auxe will show the environment vars!

wchan

you can choose which columns to show with ps (ps -eo ...)

One cool column is 'wchan' which tells you the name of the kernel function if the process is sleeping

try it:

```
ps -eo user,pid,wchan,cmd
```

★ process state ★

Here's what the letters in ps's STATE column mean:

R: running
S/D: asleep
Z: zombie
l: multithreaded
+: in the foreground

f

is for "forest". ps auxf will show you an ASCII art process tree!

pstree can display a process tree too

ps has 3 different sets of command line arguments 💔

1. UNIX (1 dash)
2. BSD (no dash)
3. GNU (2 dashes)

you can write monstrositres like:

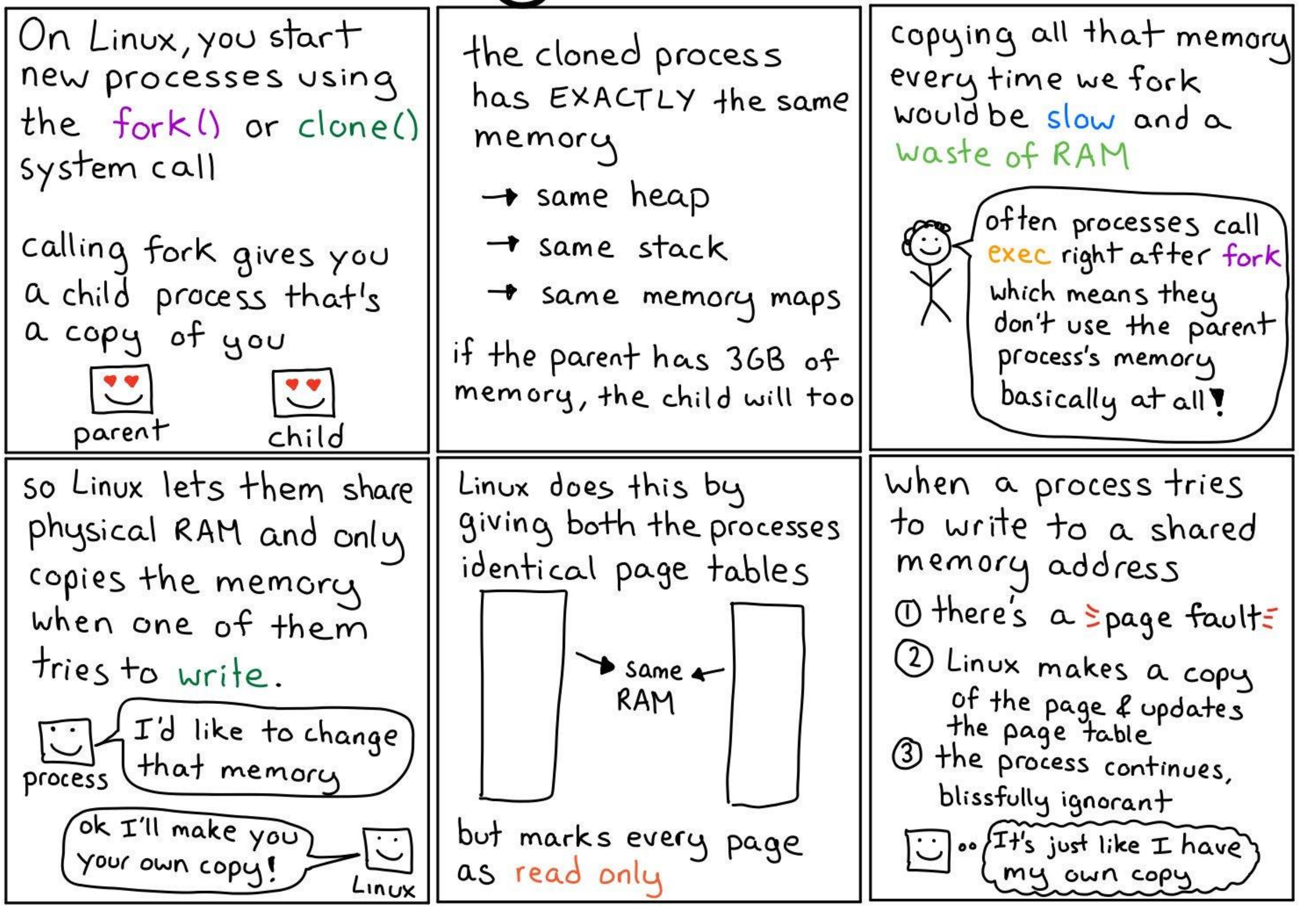
```
$ ps f -f
```

forest (BSD)

full format (UNIX)

copy on write

JULIA EVANS
@b0rk



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

calling fork gives you a child process that's a copy of you

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
ok I'll make you your own copy!
Linux

Linux does this by giving both the processes identical page tables

but 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

Copy on Write

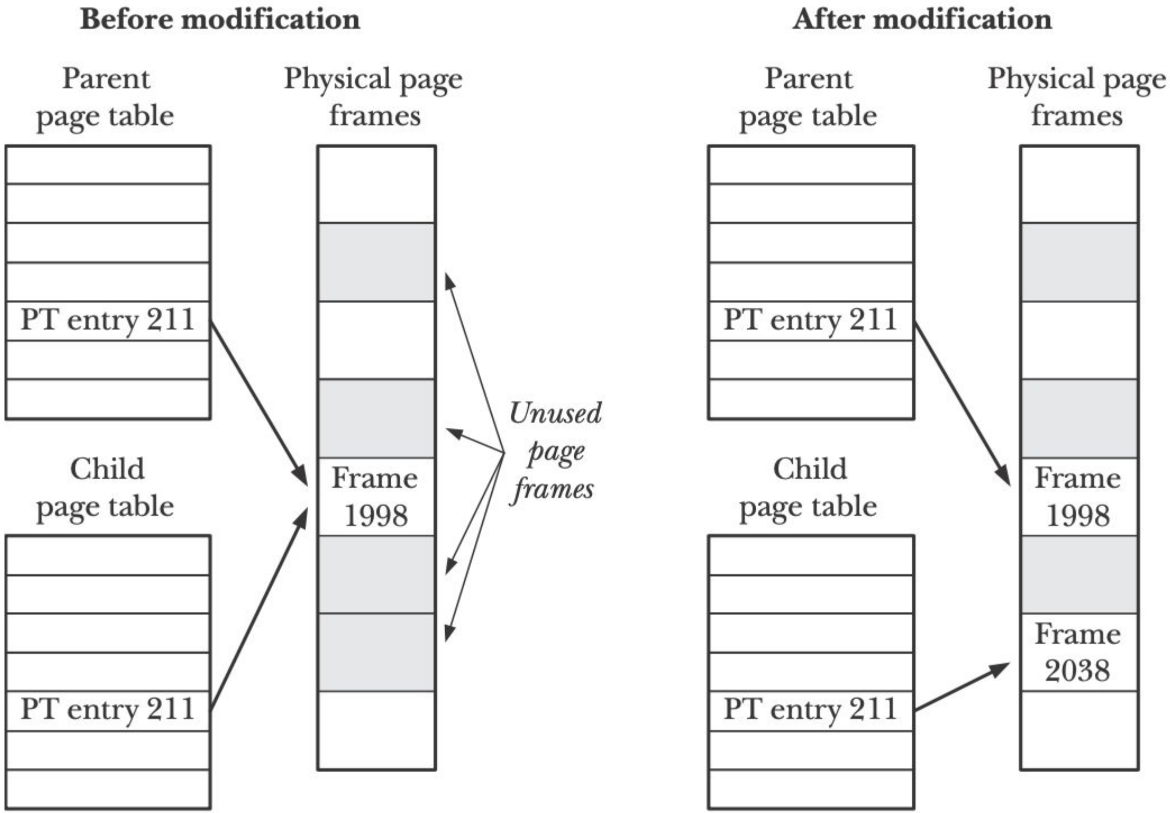


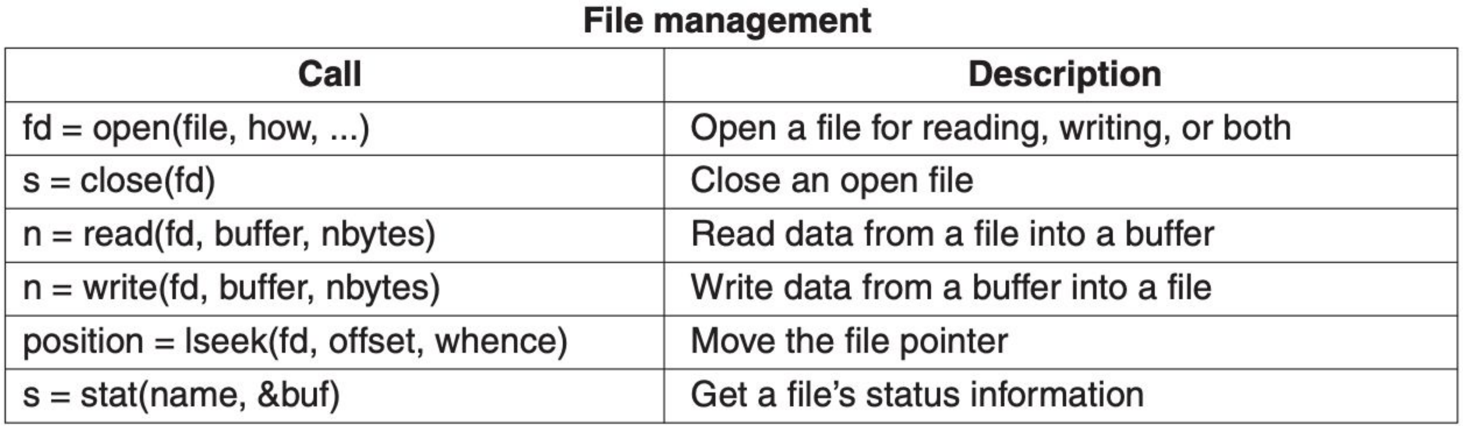
Figure 24-3: Page tables before and after modification of a shared copy-on-write page

The Linux programming interface - Kerrisk, Michael

POSIX

open read write close

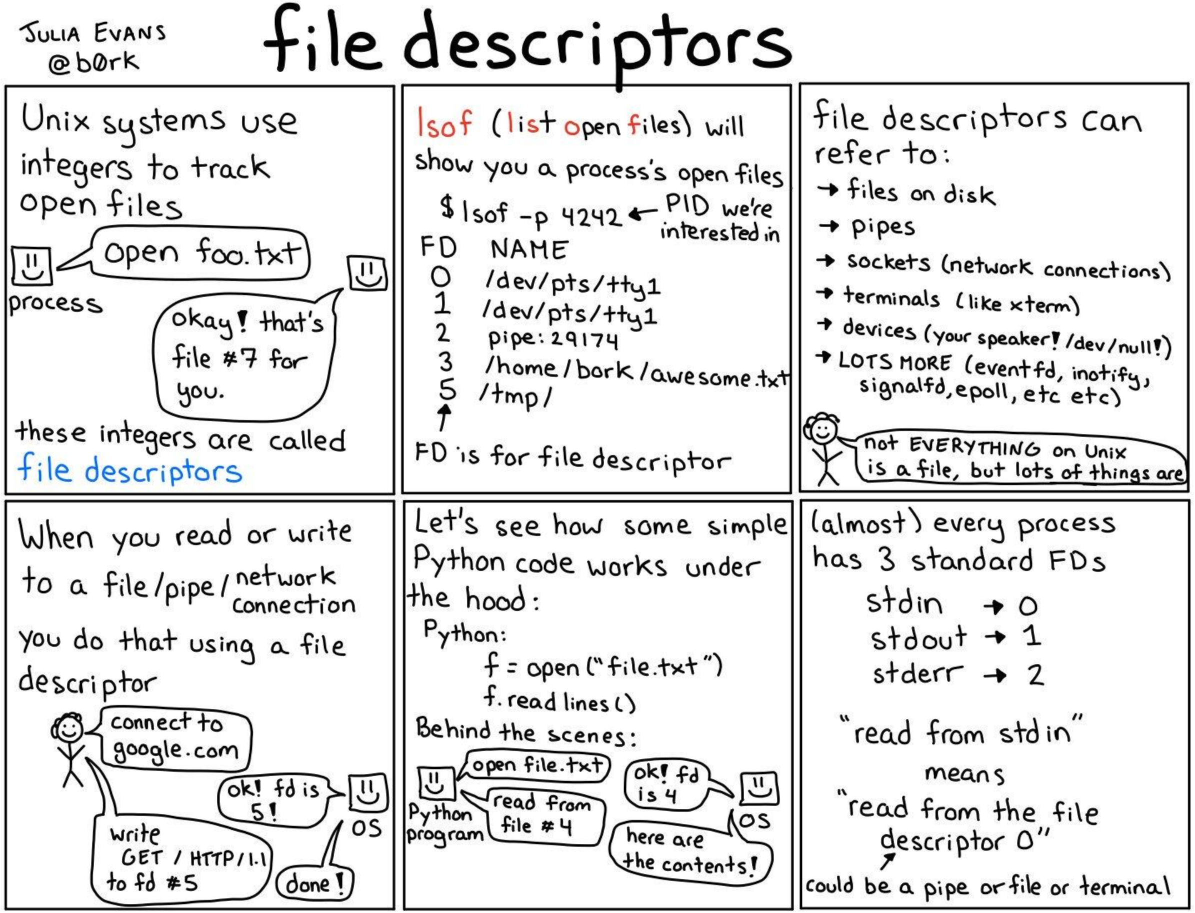
- Universalidad de I/O en UNIX
- Eficiencia read write
- Protocolo
    - End of file
    - Read 0
    - Read block



File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing, or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information

POSIX



JULIA EVANS
@b0rk

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 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

POSIX

pipe

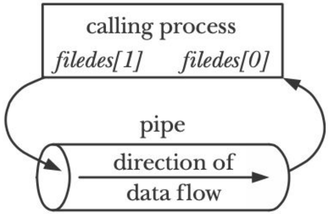
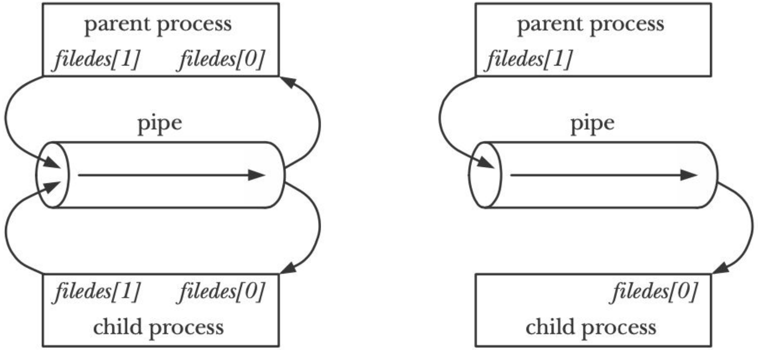


Figure 44-2: Process file descriptors after creating a pipe

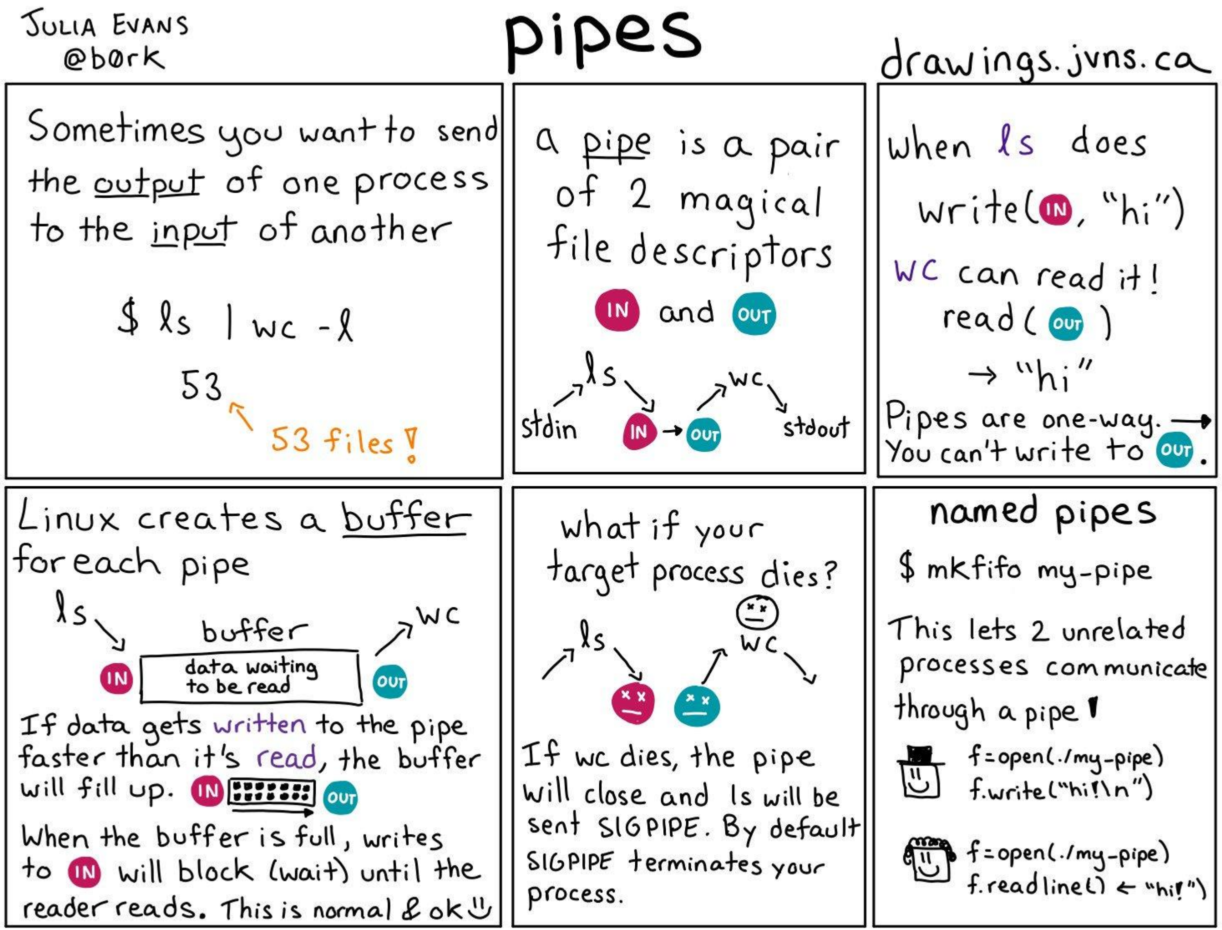


a) After *fork()*   **b) After closing unused descriptors**

Figure 44-3: Setting up a pipe to transfer data from a parent to a child

The Linux programming interface - Kerrisk, Michael

POSIX



JULIA EVANS
@b0rk

pipes

drawings.jvns.ca

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 IN and OUT

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!")
```

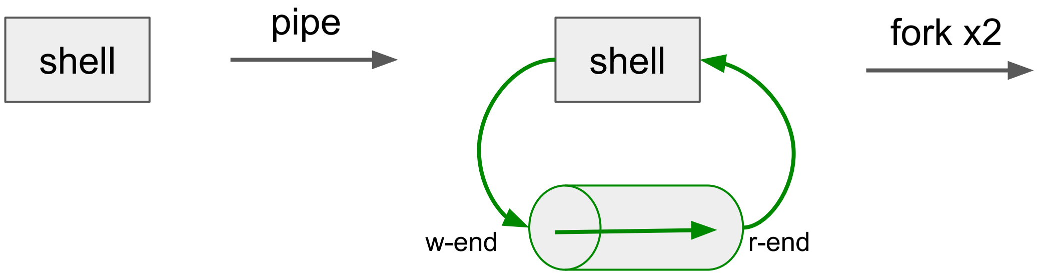
POSIX

dup

- DEMO shell, cat y strace

shell	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty

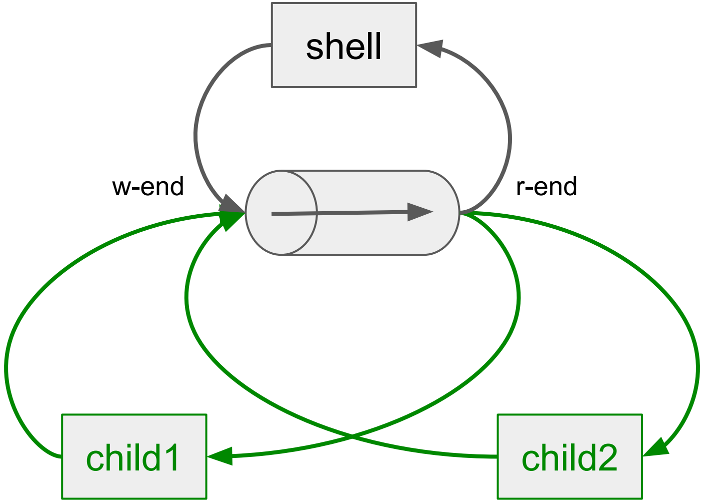
shell	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty
3	r-end
4	w-end



shell	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty
3	r-end
4	w-end

child1	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty
3	r-end
4	w-end

child2	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty
3	r-end
4	w-end

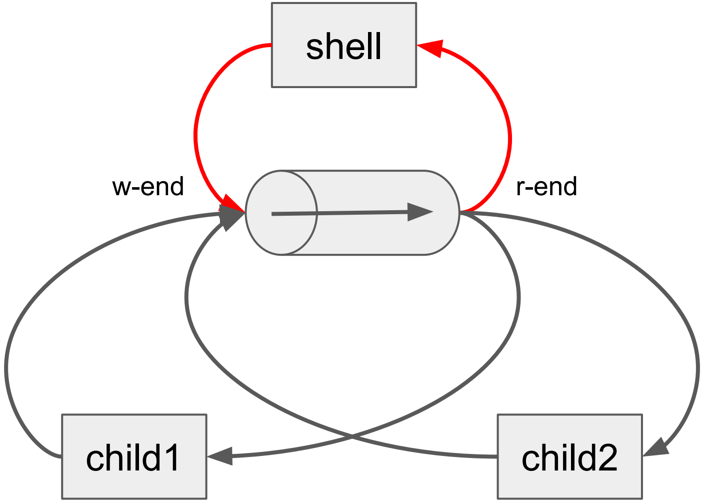


POSIX
dup

shell	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty
3	**r-end**
4	**w-end**

child1	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty
3	r-end
4	w-end

child2	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty
3	r-end
4	w-end



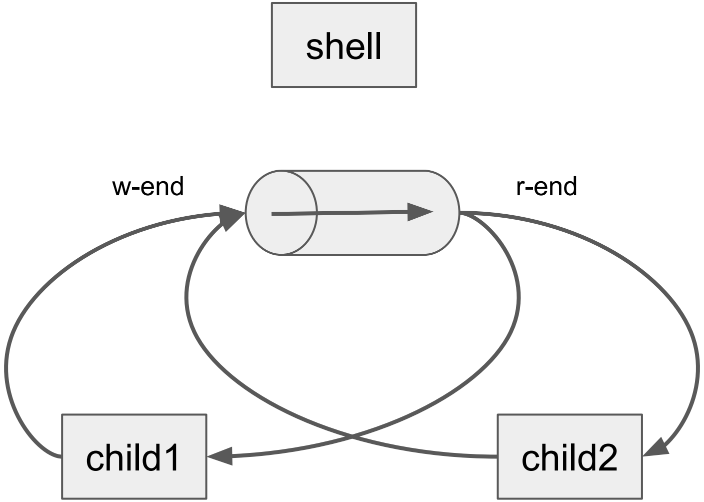
shell

close(3)
close(4)

shell	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty

child1	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty
3	r-end
4	w-end

child2	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty
3	r-end
4	w-end



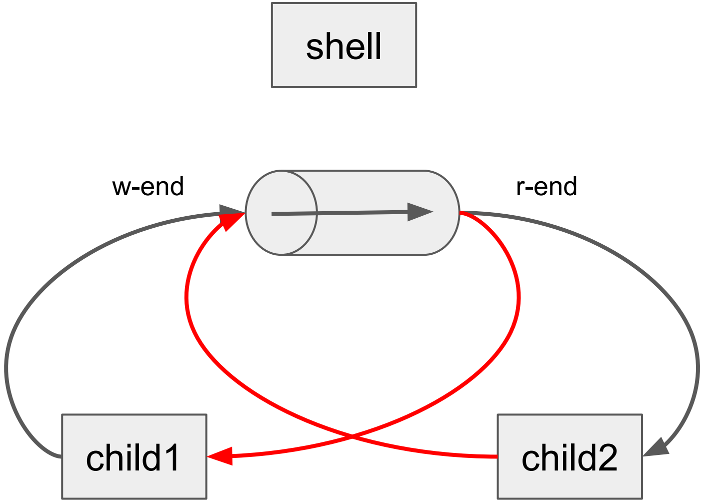
POSIX

dup

shell	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty

child1	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty
3	**r-end**
4	w-end

child2	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty
3	r-end
4	**w-end**



child1

close(3)

child2

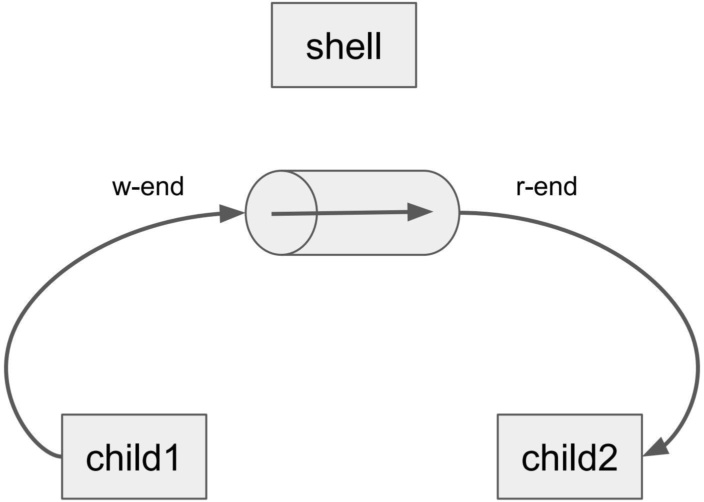
close(4)

→

shell	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty

child1	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty
4	w-end

child2	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty
3	r-end



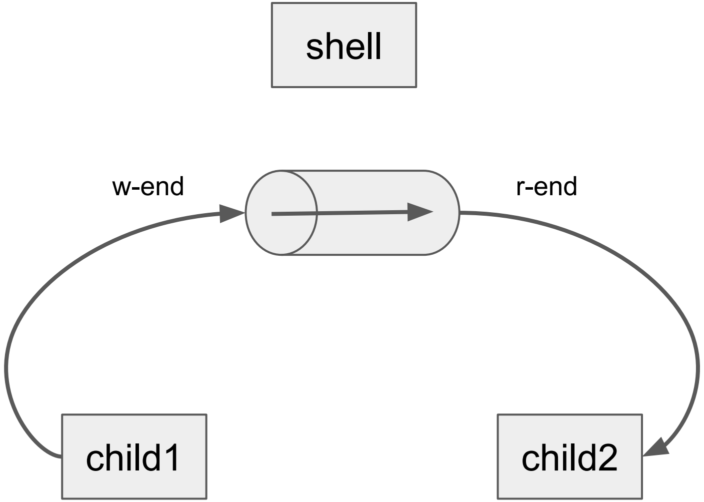
POSIX

dup

shell	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty

child1	
fd	**device**
0	/dev/tty
1	**/dev/tty**
2	/dev/tty
4	w-end

child2	
fd	**device**
0	**/dev/tty**
1	/dev/tty
2	/dev/tty
3	r-end



child1

close(1)

child2

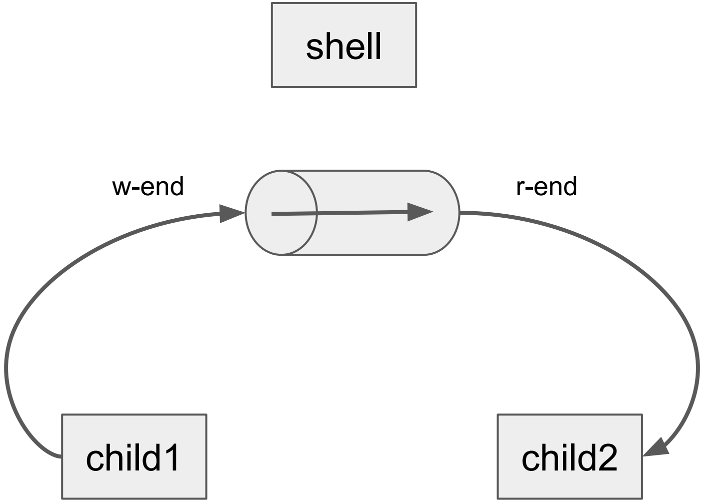
close(0)

→

shell	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty

child1	
fd	**device**
0	/dev/tty
2	/dev/tty
4	w-end

child2	
fd	**device**
1	/dev/tty
2	/dev/tty
3	r-end



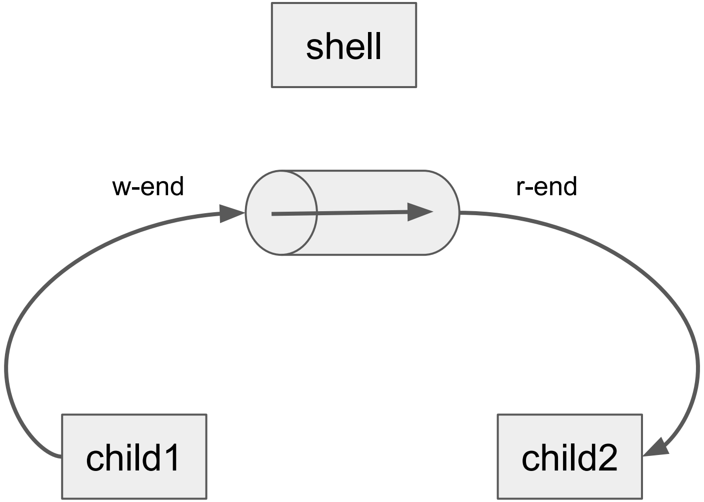
POSIX

dup

shell	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty

child1	
fd	**device**
0	/dev/tty
2	/dev/tty
4	w-end

child2	
fd	**device**
1	/dev/tty
2	/dev/tty
3	r-end



child1

dup(4)

child2

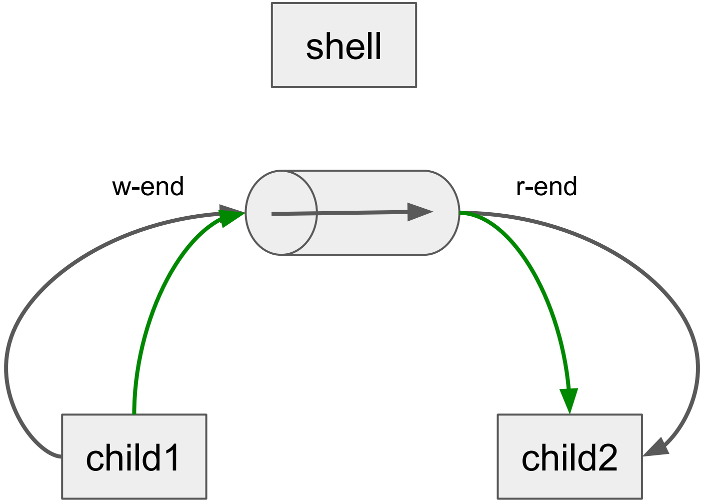
dup(3)

→

shell	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty

child1	
fd	**device**
0	/dev/tty
1	**w-end**
2	/dev/tty
4	w-end

child2	
fd	**device**
0	**r-end**
1	/dev/tty
2	/dev/tty
3	r-end



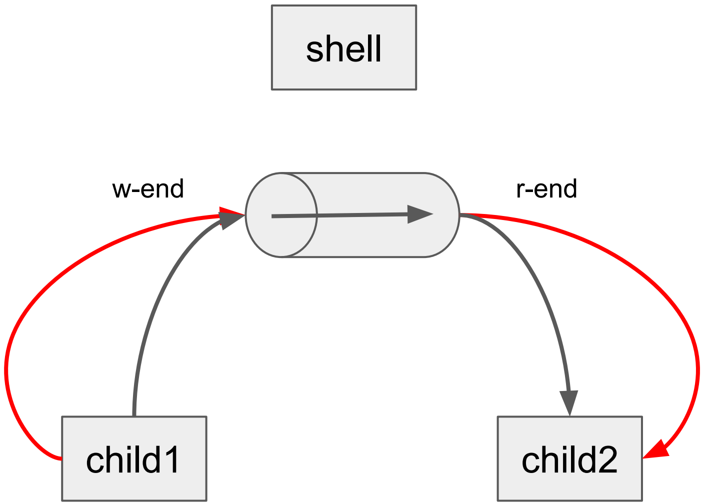
POSIX

dup

shell	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty

child1	
fd	**device**
0	/dev/tty
1	w-end
2	/dev/tty
4	**w-end**

child2	
fd	**device**
0	r-end
1	/dev/tty
2	/dev/tty
3	**r-end**



child1

close(4)

child2

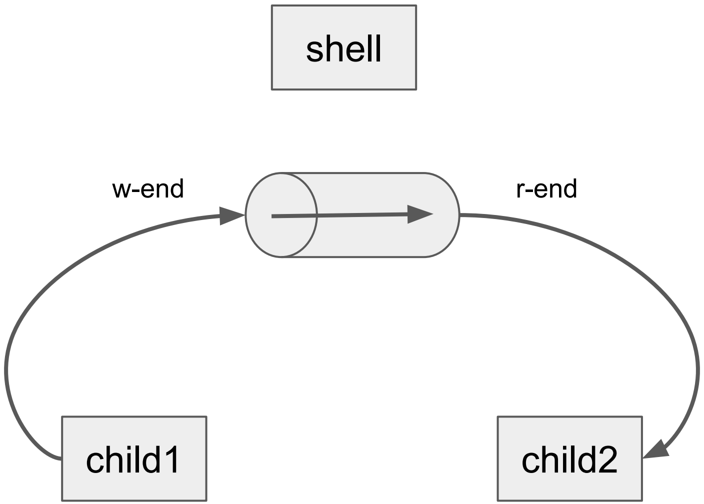
close(3)

→

shell	
fd	**device**
0	/dev/tty
1	/dev/tty
2	/dev/tty

child1	
fd	**device**
0	/dev/tty
1	w-end
2	/dev/tty

child2	
fd	**device**
0	r-end
1	/dev/tty
2	/dev/tty



POSIX

dup

```
//...
 case PIPE:
   pcmd = (struct pipecmd*)cmd;
   if(pipe(p) < 0)
     panic("pipe");
   if(fork1() == 0){
     close(1);
     dup(p[1]);
     close(p[0]);
     close(p[1]);
     runcmd(pcmd->left);
   }
   if(fork1() == 0){
     close(0);
     dup(p[0]);
     close(p[0]);
     close(p[1]);
     runcmd(pcmd->right);
   }
   close(p[0]);
   close(p[1]);
   wait();
   wait();
   break;
 //...
```

https://github.com/mit-pdos/xv6-public/blob/master/sh.c

Ejercicio 1

Escribir un programa en C que liste recursivamente el contenido de un directorio que recibe como primer y único argumento, indicar si cada elemento es un archivo o un directorio, tabular la salida por nivel de anidamiento (**`stat`** **`opendir`** **`readdir`**)

Ejemplo de la salida (solo a modo ilustrativo):

```
$./tree .
d     dir1
f     file1
d     dir3
f           file1
d           dir1
f                 file1
f     file2
```

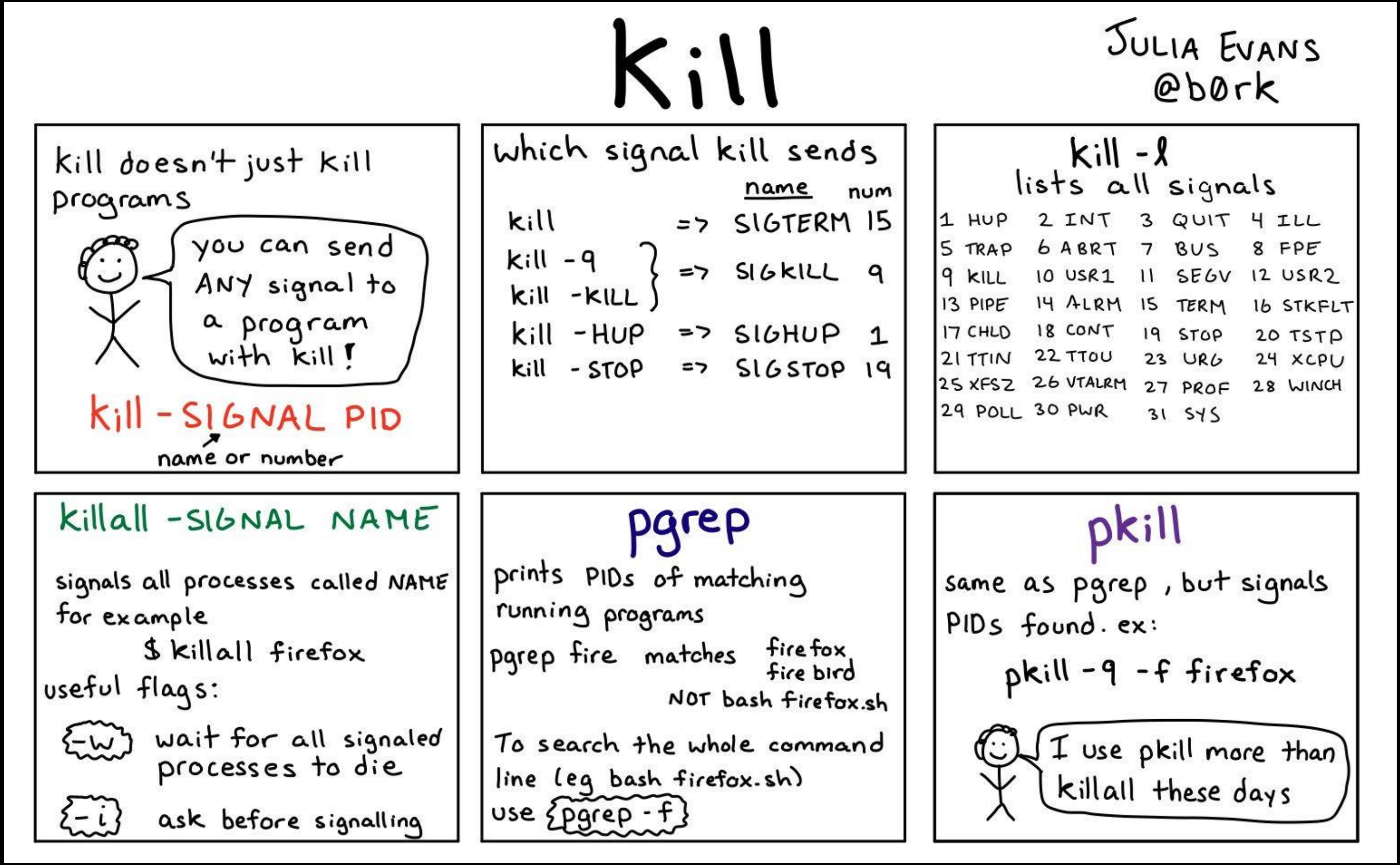
Ejercicio 2

1. Escribir un programa en C que lance 10 procesos que realicen alguna tarea que dure una cierta cantidad de tiempo considerable (por ejemplo, dormir una cantidad al azar de tiempo), el programa debe esperar a que sus procesos hijos terminen para terminar él mismo. (**fork waitpid**).

2. Hacer que cada hijo imprima su propio pid (**getpid**)

3. Separe el código de los hijos y del padre en dos unidades de compilación diferentes (**execve**)

Ejercicio 3



kill

Julia Evans
@b0rk

kill doesn't just kill programs

kill -SIGNAL PID

name or number

which signal kill sends

		name	num
kill	=>	SIGTERM	15
kill -9 kill -KILL	=>	SIGKILL	9
kill -HUP	=>	SIGHUP	1
kill -STOP	=>	SIGSTOP	19

kill -l

lists all signals

1 HUP	2 INT	3 QUIT	4 ILL
5 TRAP	6 ABRT	7 BUS	8 FPE
9 KILL	10 USR1	11 SEGV	12 USR2
13 PIPE	14 ALRM	15 TERM	16 STKFLT
17 CHLD	18 CONT	19 STOP	20 TSTP
21 TTIN	22 TTOU	23 URG	24 XCPU
25 XFSZ	26 VTALRM	27 PROF	28 WINCH
29 POLL	30 PWR	31 SYS	

killall -SIGNAL NAME

signals all processes called NAME for example

```
$ killall firefox
```

useful flags:

- -w wait for all signaled processes to die
- -i ask before signalling

pgrep

prints PIDs of matching running programs

pgrep fire matches firefox
fire bird
NOT bash firefox.sh

To search the whole command line (eg bash firefox.sh) use pgrep -f

pkill

same as pgrep, but signals PIDs found. ex:

```
pkill -9 -f firefox
```

Ejercicio 3

Lance varios procesos hijos desde un padre, haga que el padre cuelgue en un loop infinito sin hacer **`waitpid`**, mientras que sus hijos terminan instantáneamente, salvo 1 que también quedará en un loop infinito.

Para observar el estado de los procesos se recomienda ejecutar

```
$ ps ax -O "%P"
```

1. ¿Qué imagina que va a pasar con los procesos hijos que terminaron? ¿Por qué cree que pasa?
2. Mate el proceso padre (**`kill)`** ¿Qué sucede?
3. ¿Cambió algo para el proceso hijo que estaba en un loop luego de matar al padre?
4. ¿Qué es el proceso **`init`**?
5. ¿Qué pasa en los sistemas **`POSIX`** con los procesos huérfanos?

Ejercicio 4

Dados los programas dentro del repositorio ejemplos/producer-consumer y la resolución del inciso 3 del ejercicio 2:

1. Modifique la resolución del inciso 3 del ejercicio 2, de manera tal que el proceso padre ejecute *p* y *c* (**fork** y **exec**) conectando el stdout de p al stdin de c utilizando un pipe (**pipe**).
2. Modifique la resolución del inciso 3 del ejercicio 2, de manera tal que el proceso padre ejecute *p* en background. El programa *c* no participa en este ejercicio.
3. Modifique la resolución del inciso 3 del ejercicio 2, de manera tal que el proceso padre ejecute *p* y redireccione stdout a un archivo. (**close** y **open**). El programa *c* no participa en este ejercicio.
4. Modifique la resolución del inciso 3 del ejercicio 2, de manera tal que el proceso padre ejecute *c*, pero que en lugar de leer de stdin lea de un archivo. El programa *p* no participa en este ejercicio.

Ejercicio 5

1. Modifique la resolución del inciso 3 del ejercicio 2, de manera tal que el proceso padre ejecute 2 programas de forma secuencial considerando 3 escenarios posibles:
   a. INCONDICIONAL, primero ejecuta un proceso y cuando termina ejecuta el otro.
   b. AND, si el primer proceso falla (exit status != 0), NO se ejecuta el segundo
   c. OR, si el primer proceso termina satisfactoriamente (exit status == 0), NO se ejecuta el segundo

   Puede utilizar los programas *true.c* y *false.c* para hacer pruebas. (hint: man **true**, man **false**)
   (**wait**)

Ejercicio 6

1. Reflexión:
   a. ¿Fue necesario modificar p.c o c.c para resolver alguno de estos ejercicios?
   b. ¿Qué beneficios tiene que las syscalls **fork** y **execve** estén separadas y que **execve** preserve los archivos abiertos por el invocador?
   c. ¿Cómo cree que un intérprete de comandos como sh, bash, zsh, etc resuelve los siguientes comandos?
      i. `./p | ./c`
      ii. `./p &`
      iii. `./p > out`
      iv. `./c < in`
      v. `./true ; ./false`
      vi. `./false && ./true`
      vii. `./true || ./false`
      viii. `<( ) (process substitution)`
   d. Investigue cómo se resuelven estos problemas en windows.

Glosario

- Pair programming
- Copy on write
- File descriptor
- fork bomb