System calls

File descriptor

Un **descriptor de archivo** es una clave (índice) a una estructura de datos residente en el *kernel*, que contiene detalles de todos los archivos abiertos por un proceso.

Esta estructura de datos se llama "tabla de descriptores de archivos", y **cada proceso tiene la suya**.

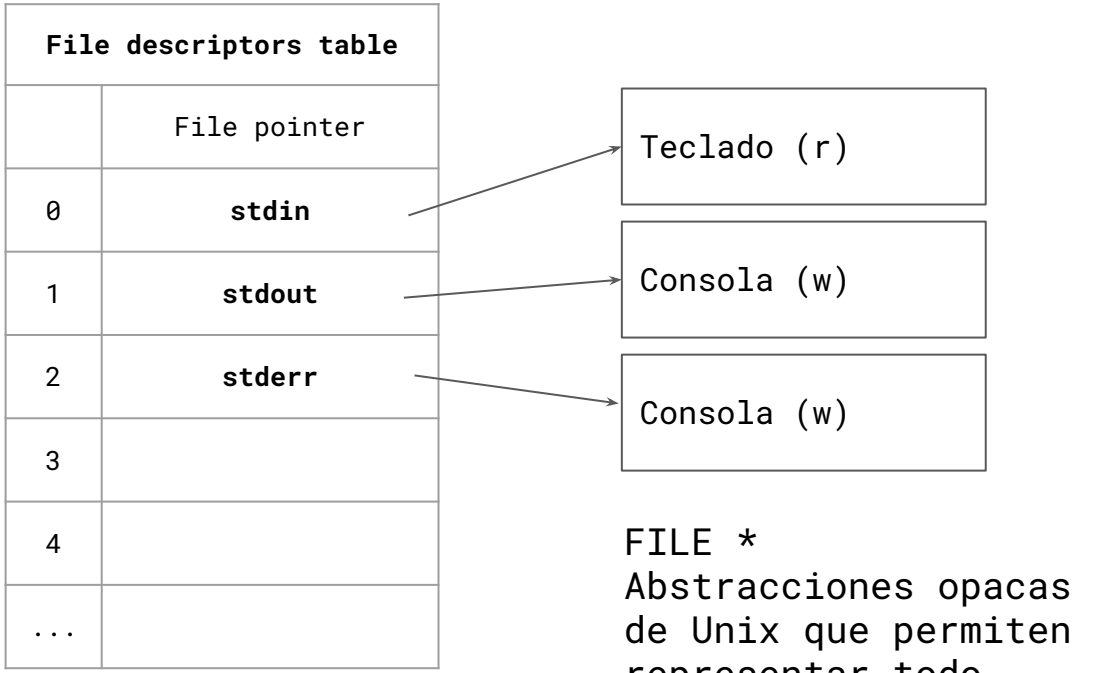
https://es.wikipedia.org/wiki/Descriptor_de_archivo

File descriptor

Un **descriptor de archivo** es una clave (índice) a una estructura de datos residente en el *kernel*, que contiene detalles de todos los archivos abiertos por un proceso.

Esta estructura de datos se llama "tabla de descriptores de archivos", y **cada proceso tiene la suya**.

Estado inicial de un proceso



FILE *
Abstracciones opacas de Unix que permiten representar todo, incluyendo dispositivos, como un archivo.

open()

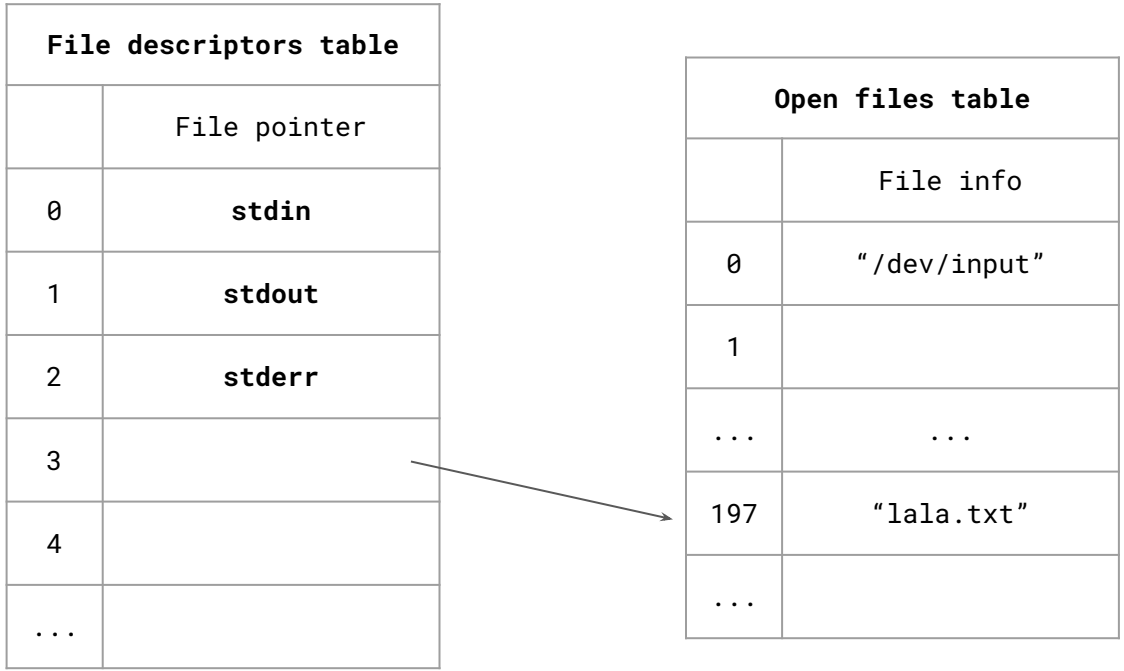
```
int open(const char *pathname, int flags, ...);
```

Open

Crea una nueva descripción de archivo abierto, es decir, una entrada en la tabla de archivos abiertos de todo el sistema.

El valor devuelto por open() es un **file pointer**, un pequeño número entero no negativo que es un índice de una entrada en la tabla de descriptores de ficheros abiertos del proceso.

Resultado de llamar a open(“lala.txt”, O_RDONLY)



https://man7.org/linux/man-pages/man2/open.2.html

Múltiples tablas de archivos en UNIX

Proceso 45

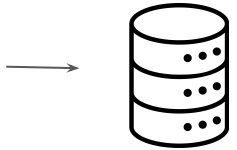
File descriptors table	
0	
1	
2	
3	
...	

Proceso 104

File descriptors table	
0	
1	
2	
3	
...	

Open files table	
	File info
0	"/dev/input" Ref count: 1, ...
1	
...	...
197	"lala.txt" Ref count: 2, ...
...	
215	"rick_and_mortyS02E01.mkv"

Inodes table	
	File metadata
...	...
3045	"lala.txt" filesize: 450B permisos: 660
...	
215	"rick_and_mortyS02E01.mkv" filesize: 1.2GB permisos: 660



Disco duro

https://man7.org/linux/man-pages/man2/open.2.html

dup2()

```
int dup2(int oldfd, int newfd);
```

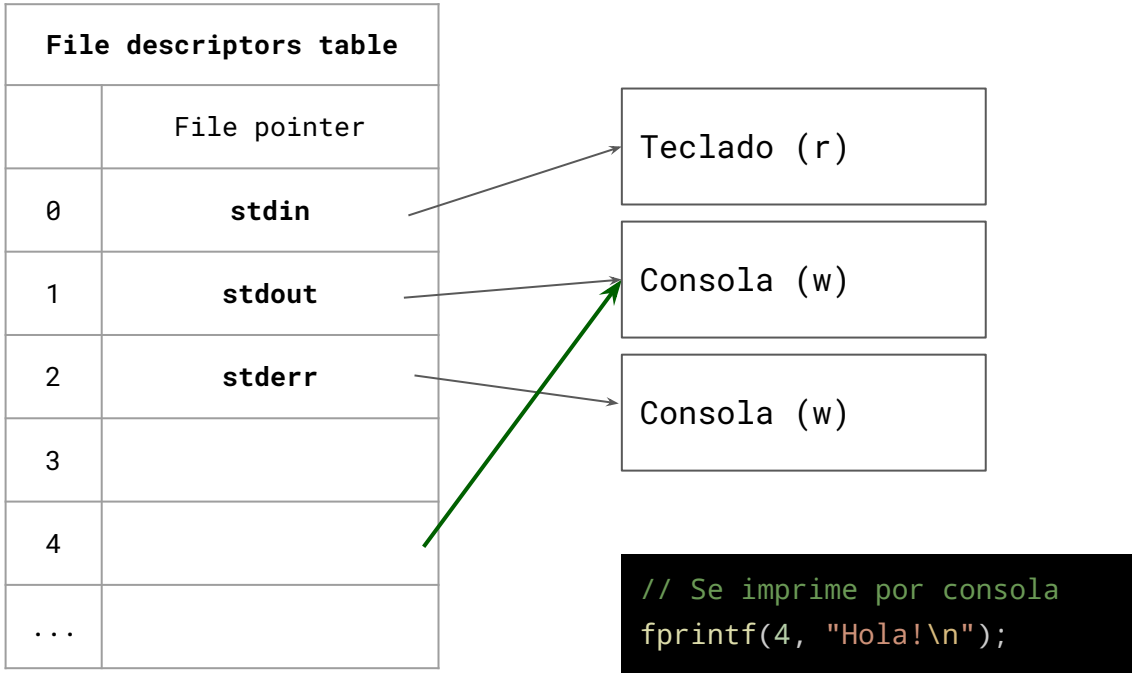
Dup2

Asigna un nuevo file descriptor que apunta al mismo file descriptor abierto que `oldfd`.

En el caso de dup2, el file descriptor `newfd` se ajusta para que ahora apunte a la misma descripción que `oldfd`.

Si el file descriptor `newfd` estaba previamente abierto, se cierra antes de ser reutilizado.

Resultado de llamar a dup2(1, 4)




```
// Se imprime por consola
fprintf(4, "Hola!\n");
```

https://man7.org/linux/man-pages/man2/dup.2.html

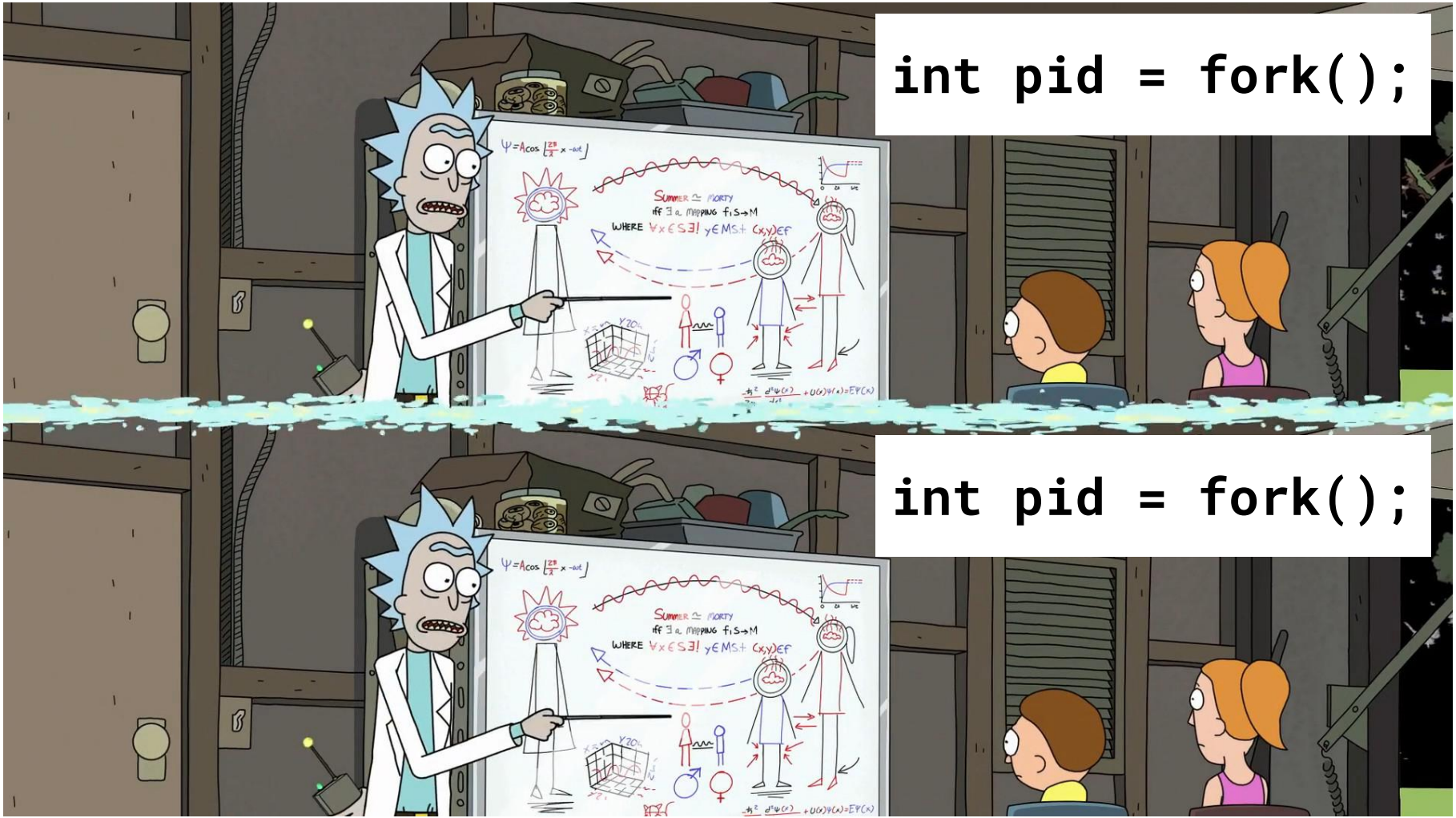
Ejercicio

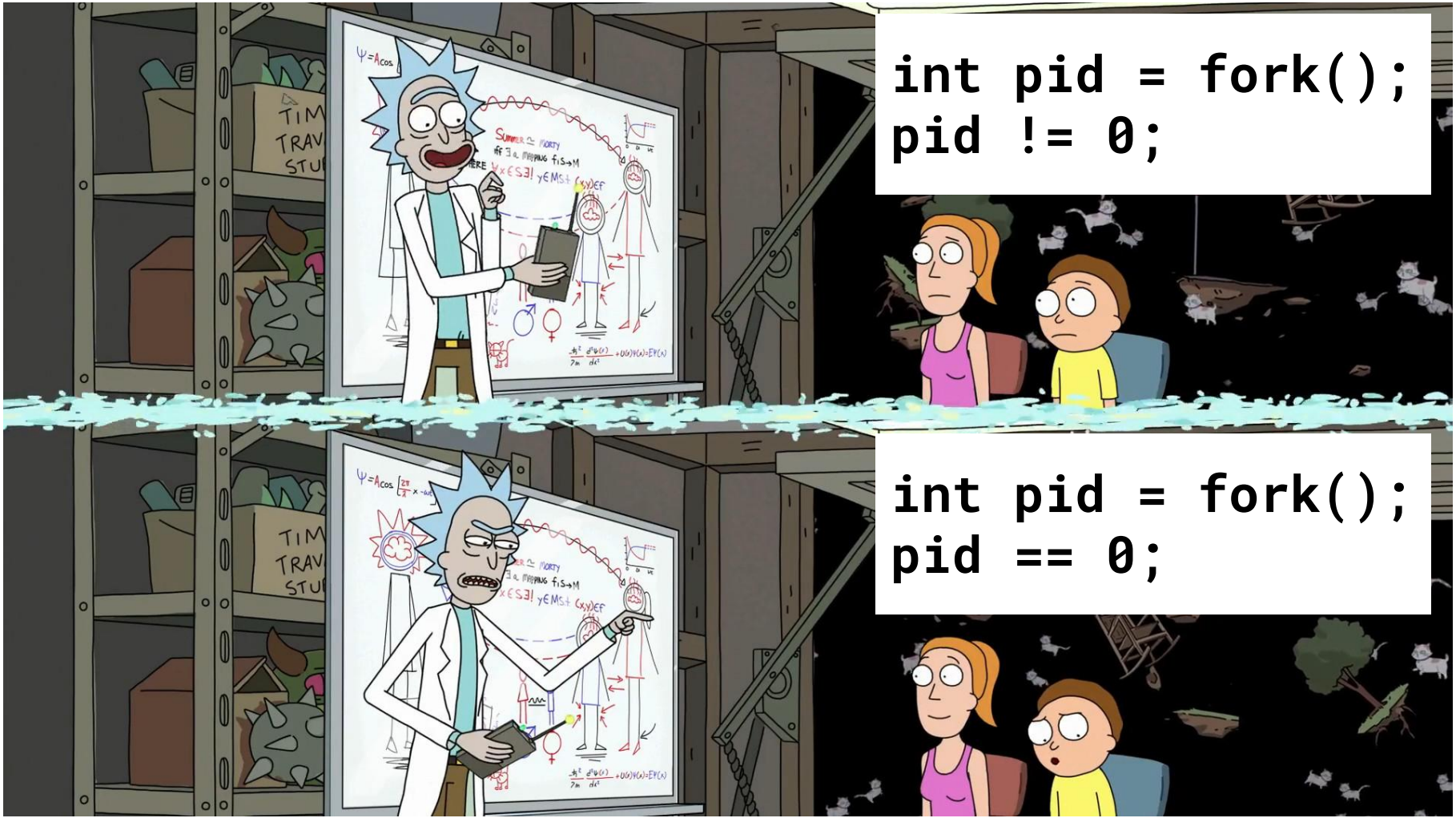
Pensar en cómo utilizar las llamadas a sistema open, dup y close para poder redireccionar la salida estándar del proceso actual.

```c
#include <stdio.h>
#include <unistd.h>

int main() {
   char *filename = "lala.txt";
   // Abrir el archivo lala.txt con permisos de escritura, creando el archivo si no existe
   // Y despues???
   // Recordar cerrar todos los file descriptors abiertos


   // Desde aquí, cualquier salida estándar se escribirá en el archivo lala.txt
   printf("Esta salida será redirigida al archivo lala.txt\n");

   return 0;
}
```

fork()

```
pid_t fork(void);
```


int pid = fork();
int pid = fork();


int pid = fork();
pid != 0;
int pid = fork();
pid == 0;

Fork

Crea un nuevo proceso duplicando el proceso original. En caso de éxito, el PID del proceso hijo se devuelve en el padre, y 0 se devuelve en el hijo.

Todo el espacio de direcciones virtual del proceso padre se replica en el proceso hijo.

El hijo **hereda copias del conjunto de file descriptors abiertos del padre**. Cada file descriptor en el hijo se refiere al mismo archivo abierto que el file descriptor correspondiente en el padre.

```c
#include  <stdio.h>

void  main(void) {
   char *filename = "lala.txt";

   fork_result = fork();
   pid = getpid();
   if (fork_result == -1) {
       printf("fork failed\n");
   } else if (fork_result == 0) {
       printf("I'm the child with pid=%d\n", pid);
   } else {
       printf(
           "I'm the parent with pid=%d and my child's pid
is=%d\n",
           pid,
           fork_result
       );
   }
}
```

Ejercicio

Utilizar fork, open, close y dup para crear dos procesos y redireccionar la salida estándar de cada uno a un archivo lala.txt, donde cada uno imprima un mensaje conteniendo su PID.

Pensar en qué orden realizar las llamadas, y asegurarse de cerrar todos los file descriptors abiertos.

```c
#include  <stdio.h>

void  main(void) {
   char *filename = "lala.txt";

   // Completar aca!

   // Desde aquí, cualquier salida estándar se escribirá en el archivo lala.txt
   printf("Esta salida será redirigida al archivo lala.txt\n");

   return 0;
}
```

execvp()

```
int execvp(const char *filename, char *const argv[]);
```


parent
execv(“chernobyl”, &args);


parent
MOSCOW
APRIL 26, 1988

Execvp

La familia de funciones exec() sustituye la imagen de proceso actual por una nueva imagen de proceso.

Ejecuta el programa apuntado por filename, que debe ser un ejecutable binario o un script.

La función no retorna en caso de éxito, y el texto, datos, bss y pila del proceso que la llama son sobrescritos por los del programa cargado.

```c
#include  <stdio.h>

void  main(void)
{
   char *cmd = "ls";
   char *argv[3];
   argv[0] = "ls";
   argv[1] = "-la";
   argv[2] = NULL;

   // Esto corre "ls -la" como si lo ejecutaramos
   // desde el bash
   execvp(cmd, argv);
   printf("There has been an error\n");
   return 1;
}
```

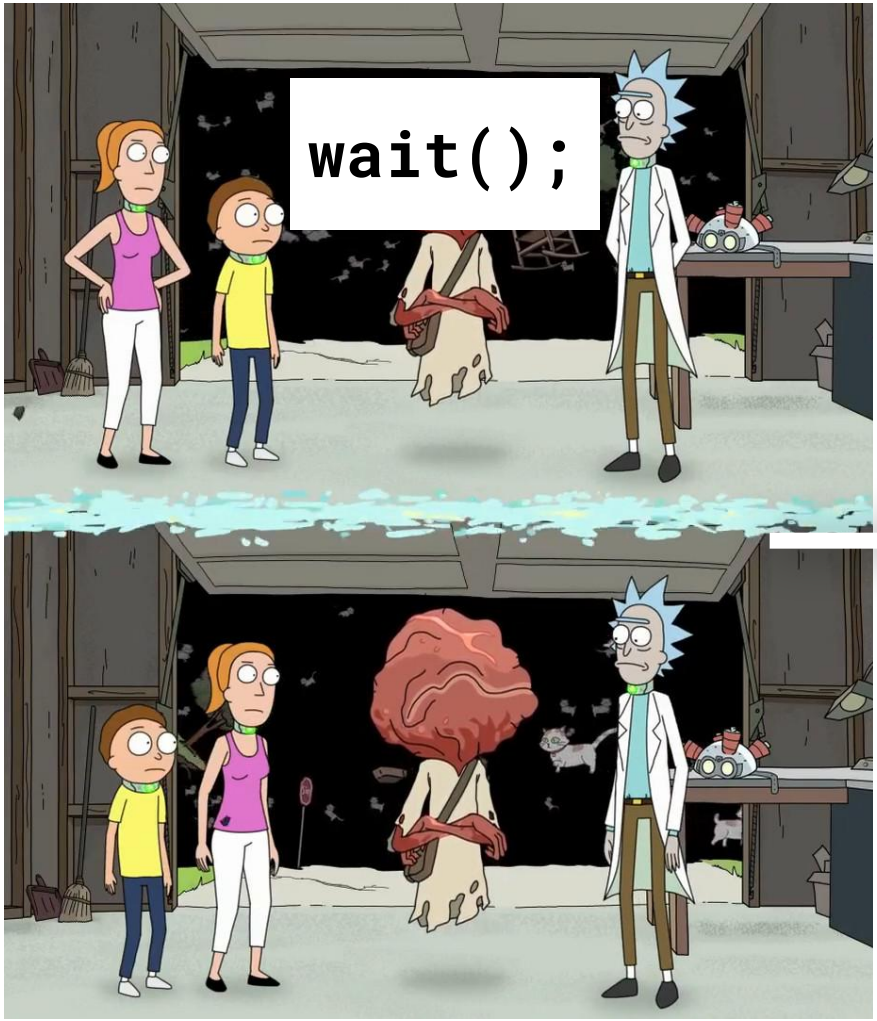
Ejercicio

1. Implementar un programa que ejecute el comando `cat /proc/cpuinfo`.
2. Implementar un programa que ejecute el mismo comando y redirija la salida estándar al archivo `cpuinfo.txt`
3.

```c
#include  <stdio.h>

void  main(void)
{
   char *cmd = "cat";
   char *argv[2];

   // Completar aca

   return 1;
}
```

wait()

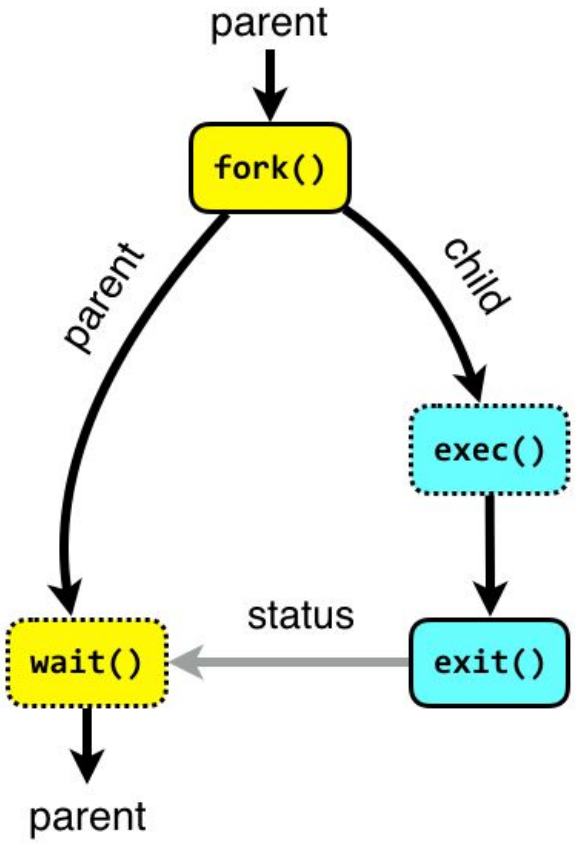
```
pid_t wait(int *wstatus);
```


wait();

Wait y waitpid

La familia de llamadas al sistema wait se utiliza para esperar cambios de estado en *un hijo* del proceso que llama, y obtener información sobre el hijo cuyo estado ha cambiado.

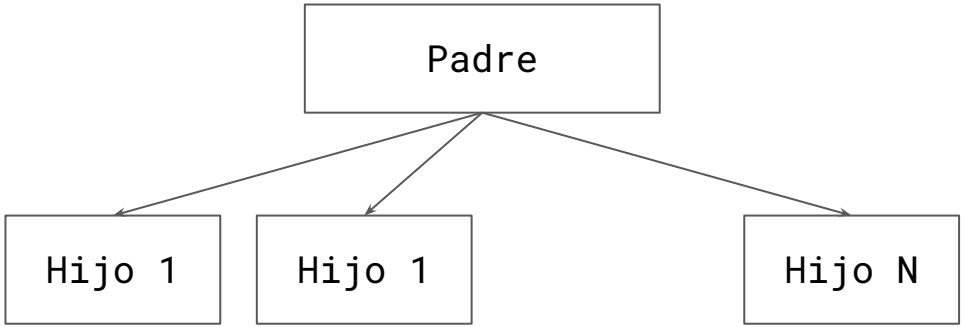
Si un hijo ya ha cambiado de estado, estas llamadas regresan inmediatamente. En caso contrario, se bloquean hasta que *un hijo* cambia de estado o un controlador de señales interrumpe la llamada.



Ejercicio

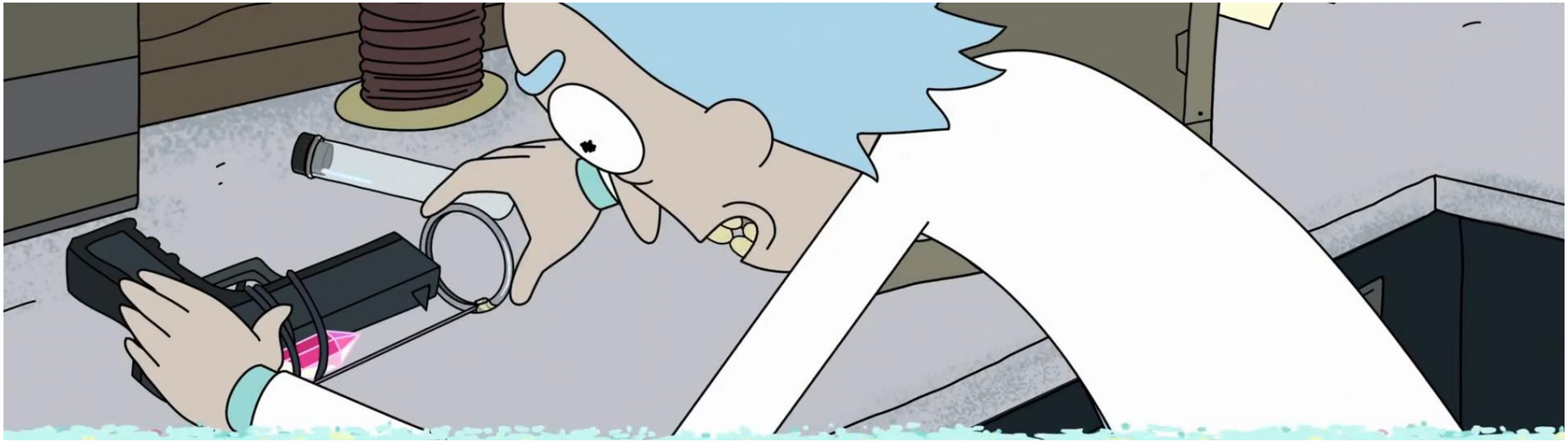
Usar todas las syscalls anteriores para ejecutar 10 programas que esperen durante 5 segundos de manera concurrente (todos al mismo tiempo), y que imprima “listo” después de que todos los procesos hijos terminan de ejecutarse.

En la vida real, reemplazaríamos el programa sleep por algo interesante como descargar archivos o renderizar video.

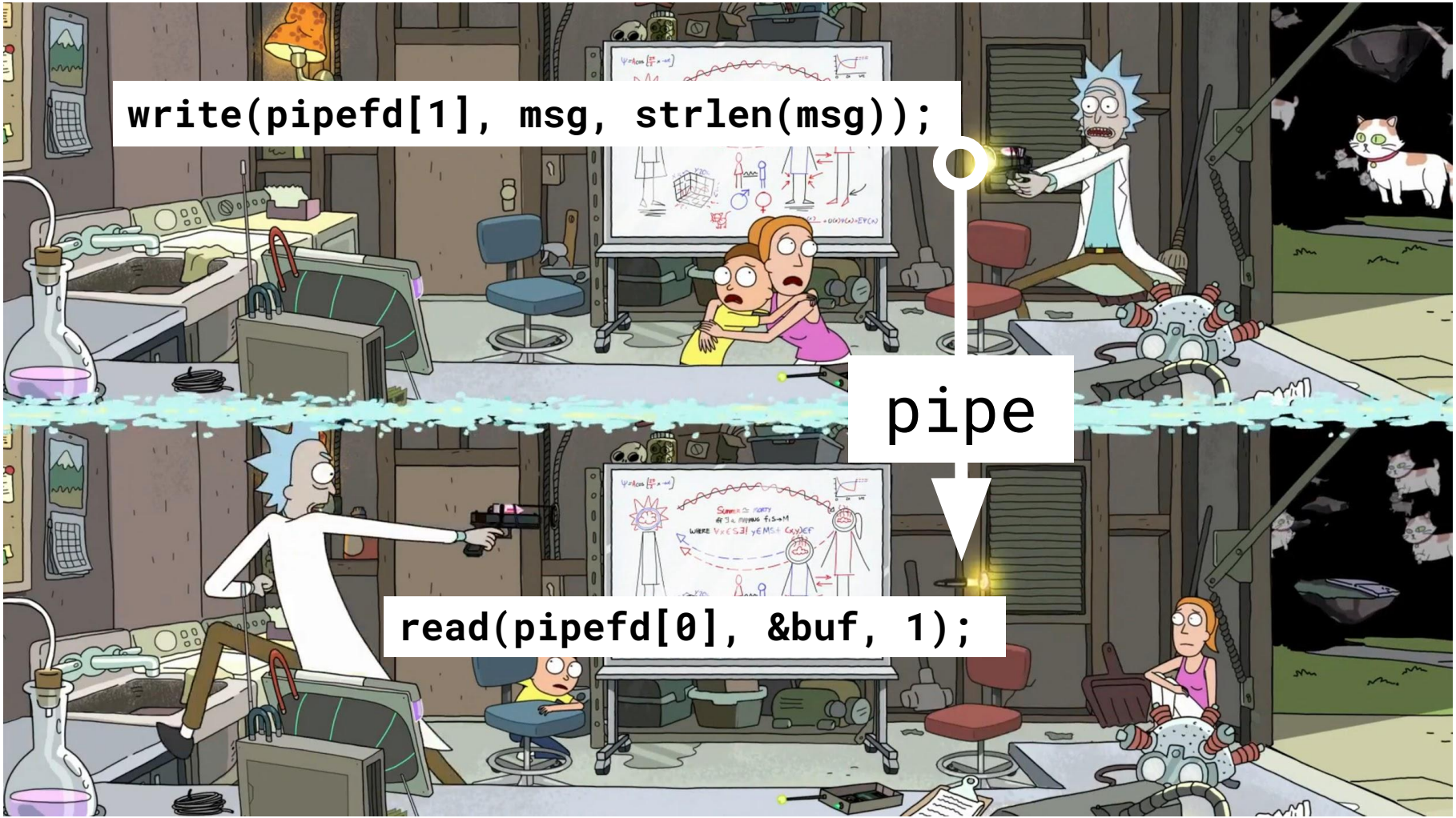


pipe()

```
int pipe(int pipefd[2]);
```



```c
int pipefd[2];
int res = pipe(pipefd);
if (res == -1) {
   exit(EXIT_FAILURE);
}
```


write(pipefd[1], msg, strlen(msg));
pipe
read(pipefd[0], &buf, 1);

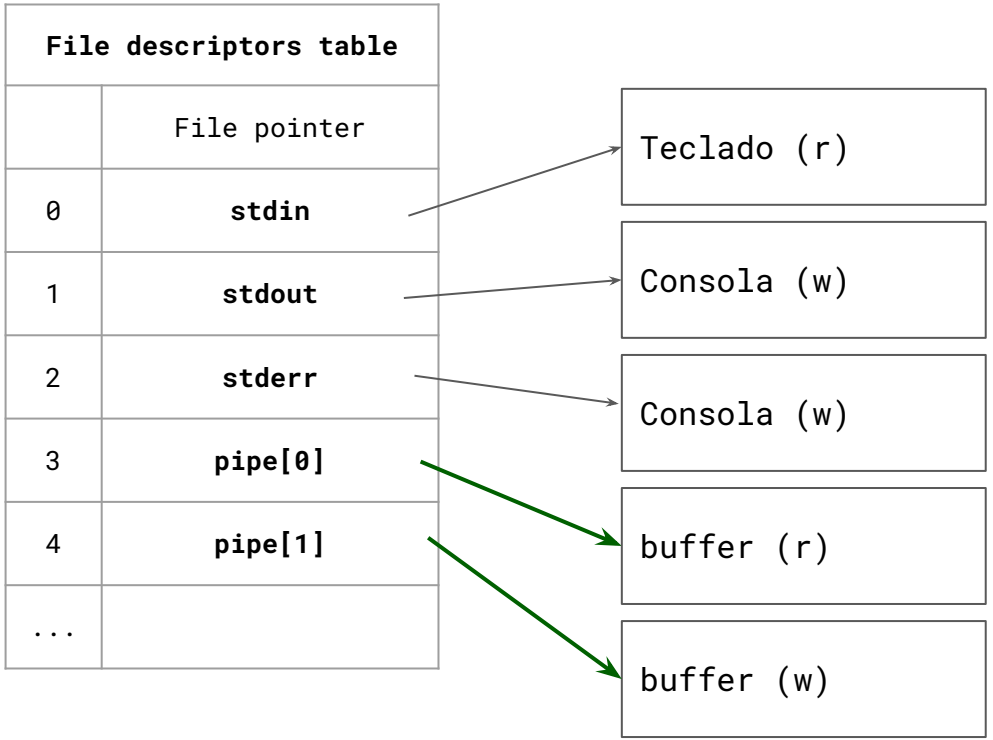
Pipe

Crea un pipe, un canal de datos unidireccional que puede utilizarse para la comunicación entre procesos. El arreglo pipefd se utiliza para devolver dos file descriptors que hacen referencia a los extremos del pipe.

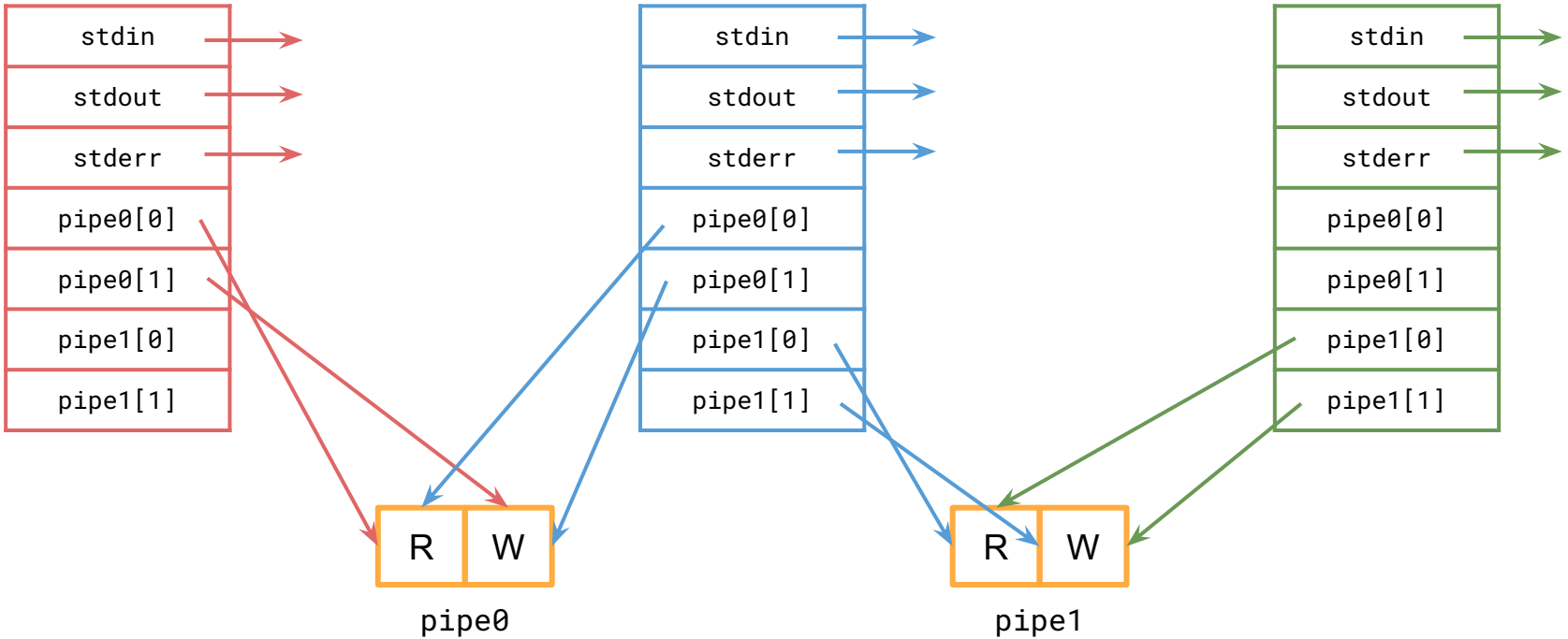
pipefd[0] se refiere al extremo de lectura del pipe.

pipefd[1] se refiere al extremo de escritura del pipe.

Resultado de llamar a pipe(pipefd)



Cómo debemos conectar y cerrar los file descriptors
para ejecutar un pipe de 3 comandos?
ls | grep “.txt” | wc -l



Recomendación:
¡lean las man pages!