**system calls explicado con rick y morty**

fork():

nuestra shell es un proceso, cuando un proceso llama a la sys call fork el sistema operativo duplica ese proceso ie. crear un nuevo proceso exactamente igual al primero pero con ciertas diferencias.

-Proceso hijo: nuevo proceso, algunas cosas las hereda del padre y otras no. Por ejemplo cada proceso tiene su propio espacio de memoria

file descriptors(si son heredados por el hijo)

-proceso original: proceso padre

fork es una syscall que no toma argumentos y devuelve un numero no negativo(PID-distinto según el proceso en el que estemos, el PID del padre contiene el PID del hijo mientras que en el hijo el PID vale 0) de esta forma podemos saber luego de llamar al fork donde estamos...si estamos en el padre o el el hijo.

Por ejemplo si estamos en el proceso hijo podemos bifurcar la ejecución y hacer una cosa totalmente distinta a lo que sta haciendo el padre

pipe():

tenemos el control entre dos procesos pero como podemos hacer para que se comuniquen entre si? hay muchas formas, una útil son los pipes:

los podemos pensar como una tubería donde en un extremo un proceso va a escribir del otro otro proceso lo va a leer..

antes del fork vamos a creas un arreglo de dos enteros: int pipefd[2] ese va a ser como nuestro pipe

int pipefd[2];

pipe(pipefd);

luego le vamos a pasar un puntero de este arreglo a la syscall quien nos va a devolver dos file descriptors: uno que apunta al extremo donde se lee y otro al extremo donde se escribe

file descriptor:

cada vez que un proceso quiere leer y escribir un archivo primero lo tiene que abrir, al hacerlo con otra syscall el so devuelve un filedescriptor que un numero que representa un archivo abierto por ese proceso.

Asi que luego de llamar a pipe tenemos dos file descriptors uno para leer y otro para escribir (comosi fueran dos archivos) porque como sabemos en unix todo es un archivo, si no fallo

ya tenemos la lista p ser usada

el indice uno del arreglo es el extremo para escribir mientras que el indice cero es el para leer

si queremos que el padre le pase datos al hijo, el padre tendra que :

write(pipe[1], msg, strlen(msg)));

mientras que el hijo va a tener que :

read(pipefd[0], &buf, 1);

dup(): -> int dup(int oldfd);

-> int dup2(int oldfd, int newfd);

es para redirigir las salidas y las entradas de los procesos

en unix todo es un archivo, y la entrada estandard y la salida estandard tmb son procesos

Todos los procesos tienen 3 file descriptors por defecto:

-stdin(standard input):-> 0

-stdout(standard output): -> 1

-stderr(standard error): -> 2

cuando hacemos printf(msg); lo que esta pasando en realidad es que se esta escribiendo el archivo:

write(1, msg, strlen(msg));

cuando hacemos scanf(“%s”, buf);

lo que estamos haciendo es leer lo que hay en un archivo especial que es apuntado por un filedescriptor cero

read(0, buf, count);

si queremos redirigir las entradas y salidas estandard tenemos que cambiar a donde apuntan el filedescriptor cero y el 1, de la siguiente manera:

dup(): hay dos versiones de dup

dup a secas recibe un argumento que es un file descriptor y trae una copia dee ese file descriptor y lo guarda en el primer espacio disponible que encuentre

en cambio dup2(1, 2) recibe dos argumentos el old file descriptor y el new file descriptor y loq eu hace es copiar el viejo pero lo guarda en el new file descriptor y si habia un arcivo abierto lo va a cerrar entonces ahora el old file descriptor y el new file descriptor apuntan al mismo archivo y son intercambiables, redirigimos todo el error estandard a la salida estandard

cuando hacemos el fork() el proceso hijo hereda una copia del filedescriptor del padre ambos apuntan al mismo lugar, ie si el hijo hace un printo es como si el padre lo hubiera hecho ya que los dos estan apuntando al mismo lugar al mismo archivo

de esta forma podemos conectar

exec(): -> int execvp(const char \*file, char \*const argv[]);

el proceso hijo de la accoin es que termina ejecutando la accion en si, para eso tenemos esta syscall

reemplaza la imagen del proceso actual por uno nuevo ie,cambia lo que se esta ejecutando por otra cosa.

la que vamos a usar:

execvp(const char \*file, char \*const argv[]);

todo el codigo que este despues del execvp no se va a ejecutar a menos que ocurra un error

proceso:

fork()

exec()

fork()

exec()

es una idea muy fuerte que por ejemplo dice que hay un proceso padre de todos:

init: PID =1 ejecutado por el so

wait(): -> pid\_t wait(int \*status);

cuando ejecutamos la shell espera hasta que el comando termine para terminar su ejecución, a menos que le pasemos algo antes:

podemos hacer que la ejecución de un proceso se suspenda hasta que alguno de sus hijos cambie su estado.

una vez que el proceso hijo hace un return el so notifica al proceso padre y continua con su ejecución