



Sistemas Operativos Fisop 2024

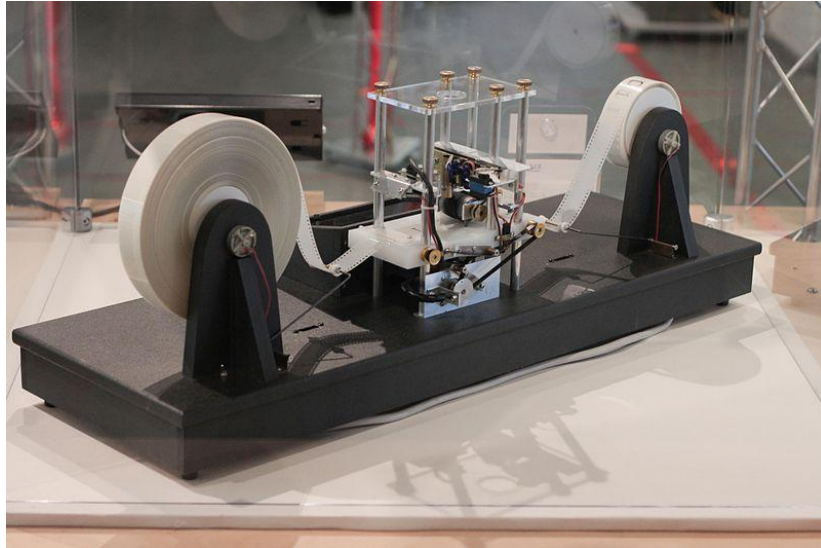
Introducción



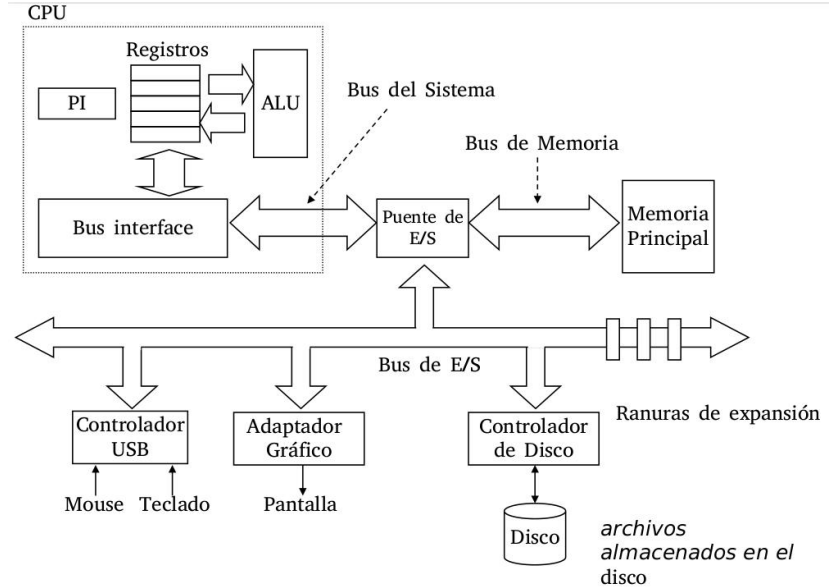
¿Qué es un Sistema Operativo?

Si te digo sistema operativo que es en lo primero que piensas?

Alan Turing



John Von Neuman



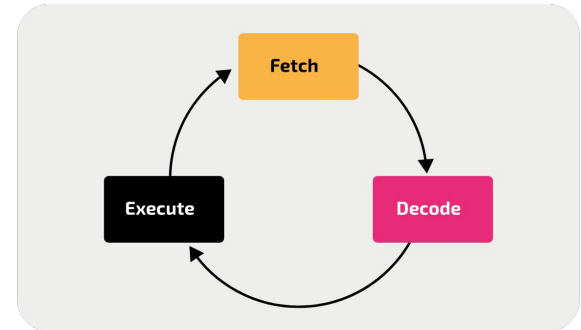
¿Qué pasa cuando un programa se ejecuta?

El procesador busca en la memoria la instrucción a ser ejecutada (**fetch**).

La decodifica (**decode**)

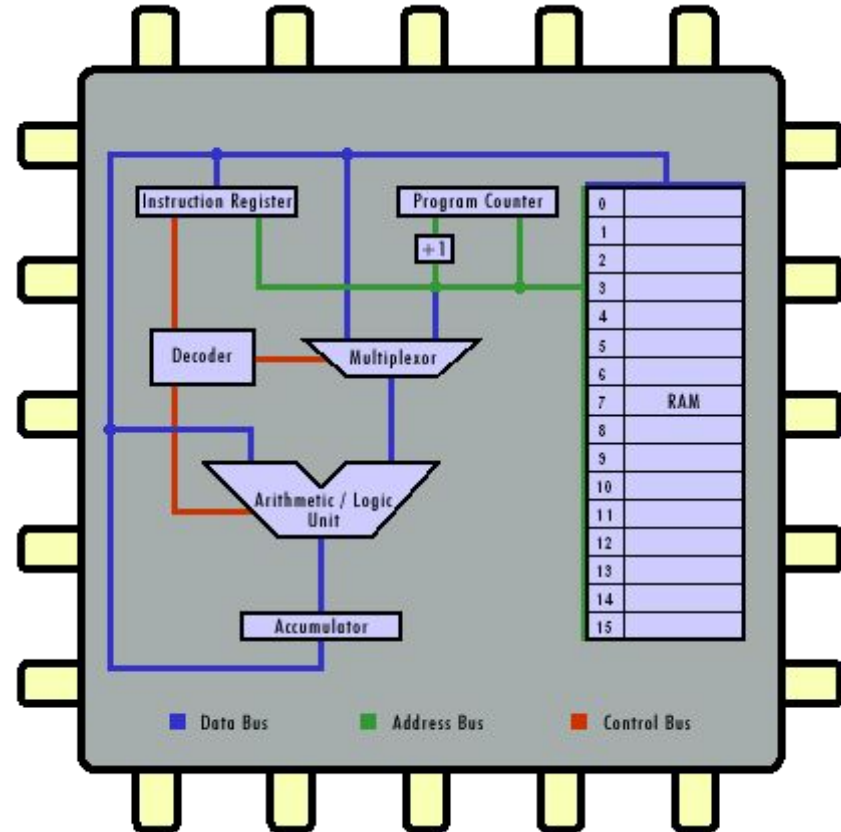
Finalmente la ejecuta (**execute**)

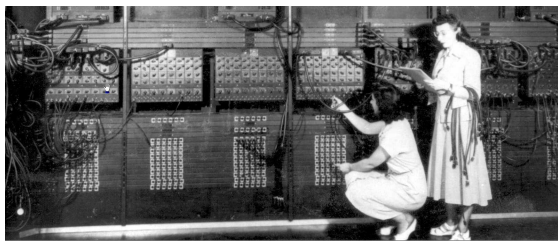
fetch-decode-execute



Arquitectura de Von Neumann

Básicamente
describimos una
forma muy simple
de la arquitectura
de Von Neumann.





Eniac



IBM 7094



IBM 370



Baby Manchester



IBM 360



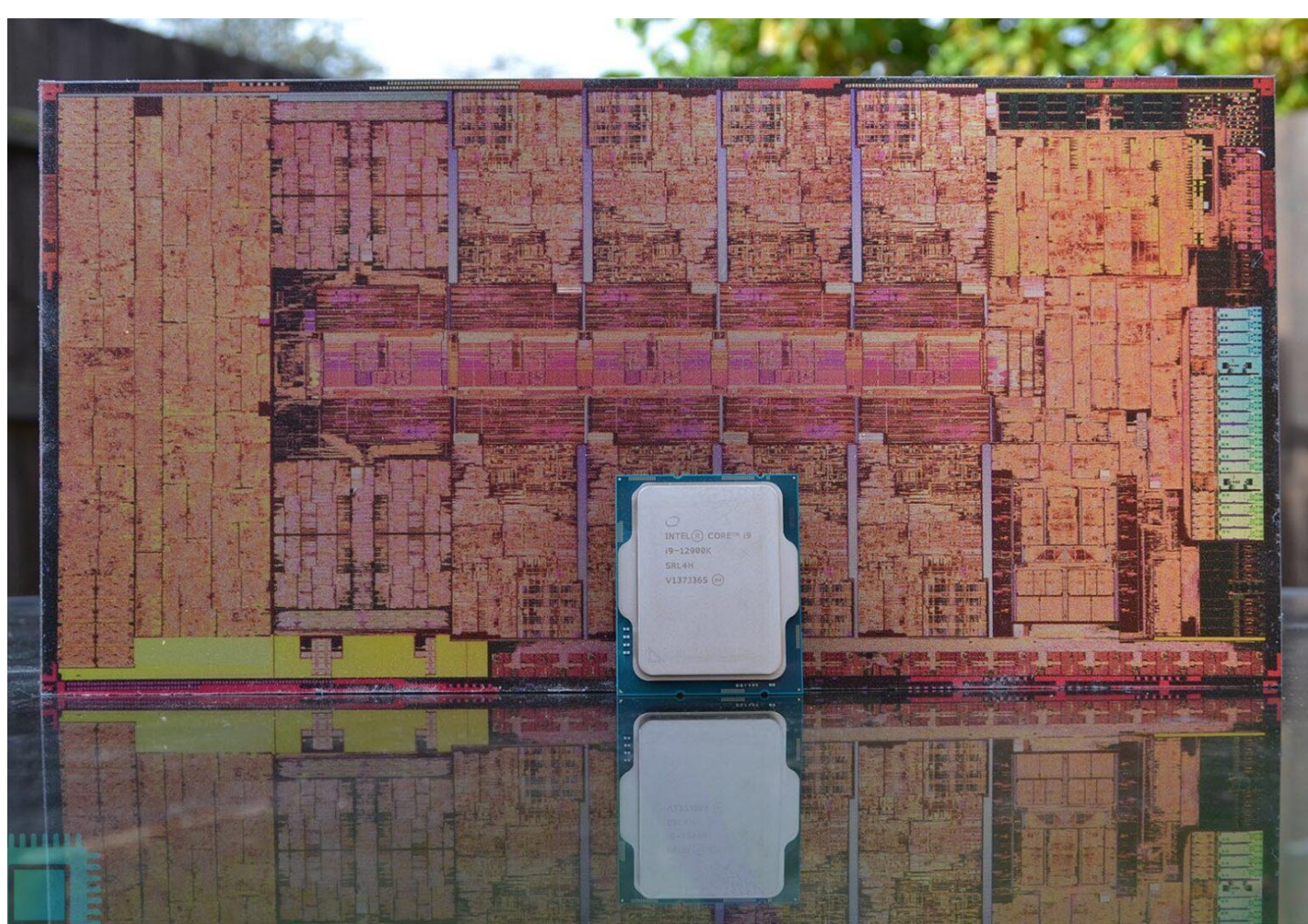
IBM 1401



IBM 390

12th Generation Intel® Core™ i9 Processors

... 8 "Golden Cove"
P-cores and 8
"Gracemont" E-cores.
The E-cores are spread
across two 4-core
"E-core Clusters."



¿Qué es un Sistema Operativo?



El rol de un sistema operativo es el de compartir una computadora entre varios programas de forma tal de proveer un conjunto de servicios útiles de los que el hardware por sí mismo expone.

¿Qué es un Sistema Operativo?



Un sistema operativo maneja y abstrae al hardware de bajo nivel, de forma tal que a un procesador de texto no le importe el tipo de disco que se está utilizando.

¿Qué es un Sistema Operativo?



Un sistema operativo provee servicios a los programas de usuario mediante una interfaz.

Diseñar una buena interfaz es toda una cuestión.

Tensión entre simplicidad vs complejidad de la interfaz

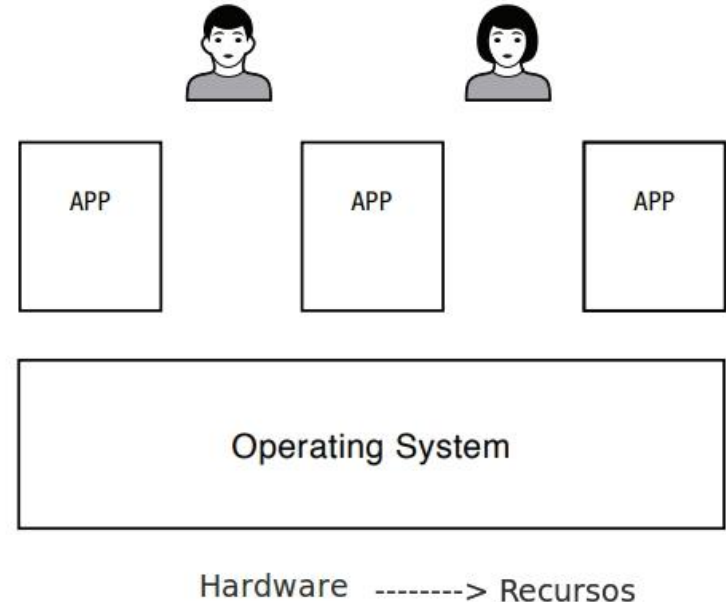
¿Qué es un Sistema Operativo?



Un **Sistema Operativo (OS)** es la **capa de software** que **maneja los recursos de una computadora** para sus **usuarios** y sus **aplicaciones**. [DAH]

¿Qué es un Sistema Operativo?

En un sistema operativo de propósito general, los **usuarios** interactúan con **aplicaciones**, estas aplicaciones se ejecutan en un **ambiente que es proporcionado** por el sistema operativo. A su vez el sistema operativo hace de **mediador** para tener acceso al **hardware del equipo**.



¿Qué es un Sistema Operativo?



Un sistema operativo es el **software** encargado de hacer que la **ejecución de los programas parezca algo fácil**. La forma principal para llegar a lograr esto es mediante el concepto de **virtualización**. Esto es, el sistema operativo toma un recurso físico (la memoria, el procesador, un disco) y lo transforma en algo virtual más general, poderoso, fácil de usar.

Virtualización



¿Cómo se logra que un recurso único pueda ser utilizado por varias entidades?

Esto es posible mediante la virtualización.

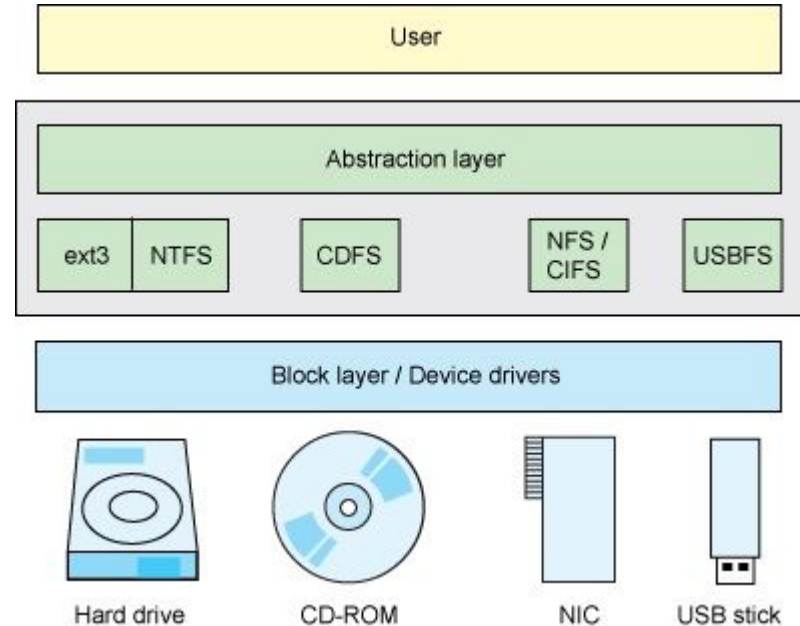
Virtualización = Abstracción

Virtualización



Virtualización: que se virtualiza (ej:src)

- La CPU
- La Memoria
- La Persistencia
- Comunicacion
- Ejecución



Funciones de un S.O.



Referee: Un OS gestiona recursos compartidos entre diferentes aplicaciones, que están ejecutándose en la misma máquina física.

Ilusionista: Un OS debe proveer una abstracción del hardware para simplificar el diseño de aplicaciones.

Pegamento: Un OS debe proveer una serie de servicios comunes que faciliten un mecanismo que permita compartir, por ejemplo, información entre las aplicaciones.... “Cut & Paste” por ejemplo ... este mecanismo es uniforme en todo el sistema.

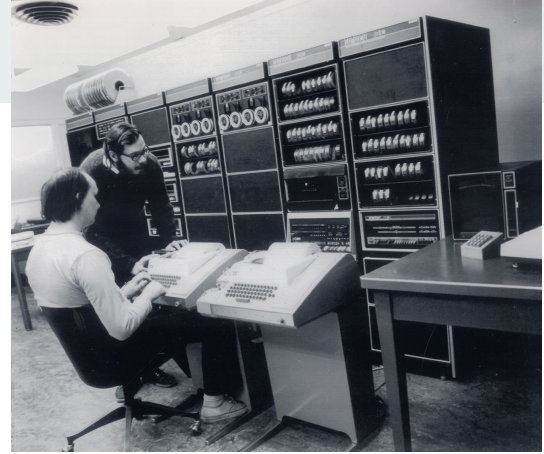
Pegamento



Uniformidad!!

- API general (Application Program Interface)
- Library Call
- System Call

Unix



- “E” sistema operativo.
- Primera version 1969.
- Tiene todo lo que un OS tiene que tener.
- Licencia Paga.



Linux:Philosophy

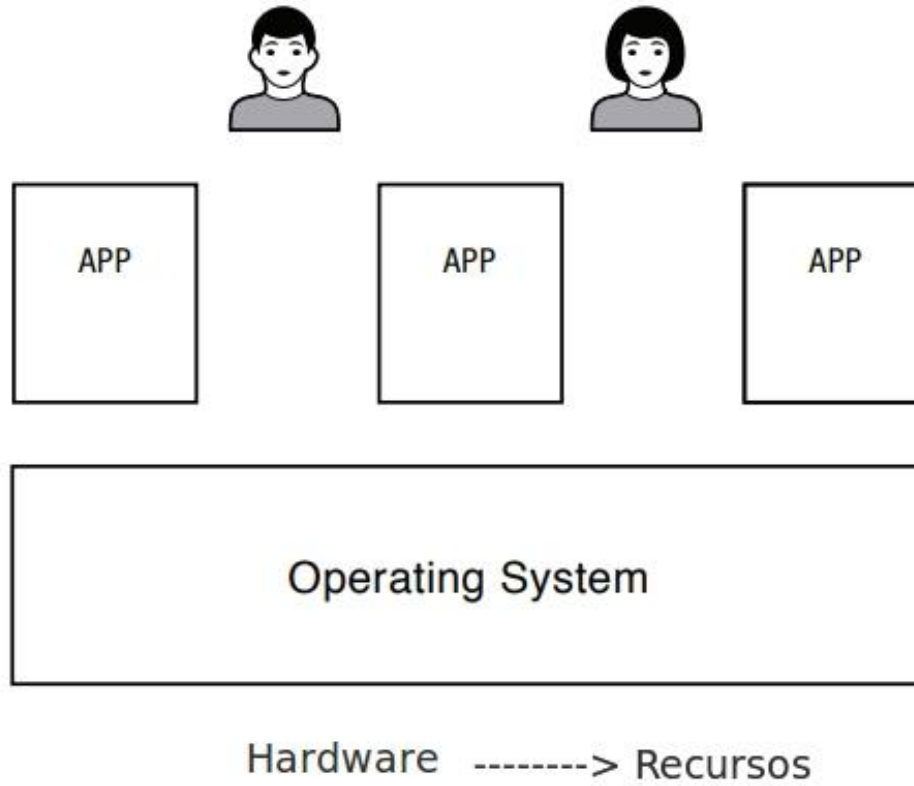
Haz que cada programa haga una cosa bien. Para hacer un nuevo trabajo, construye de nuevo en lugar de complicar los viejos programas añadiendo nuevas "características".

Espere que la salida de cada programa se convierta en la entrada de otro programa aún desconocido. No llene la salida con información extraña. Evite los formatos de entrada binarios o en columnas. No insista en la entrada interactiva.

Diseña y construye software, incluso sistemas operativos, para ser probados temprano, idealmente en semanas. No duden en tirar las partes torpes y reconstruirlas.

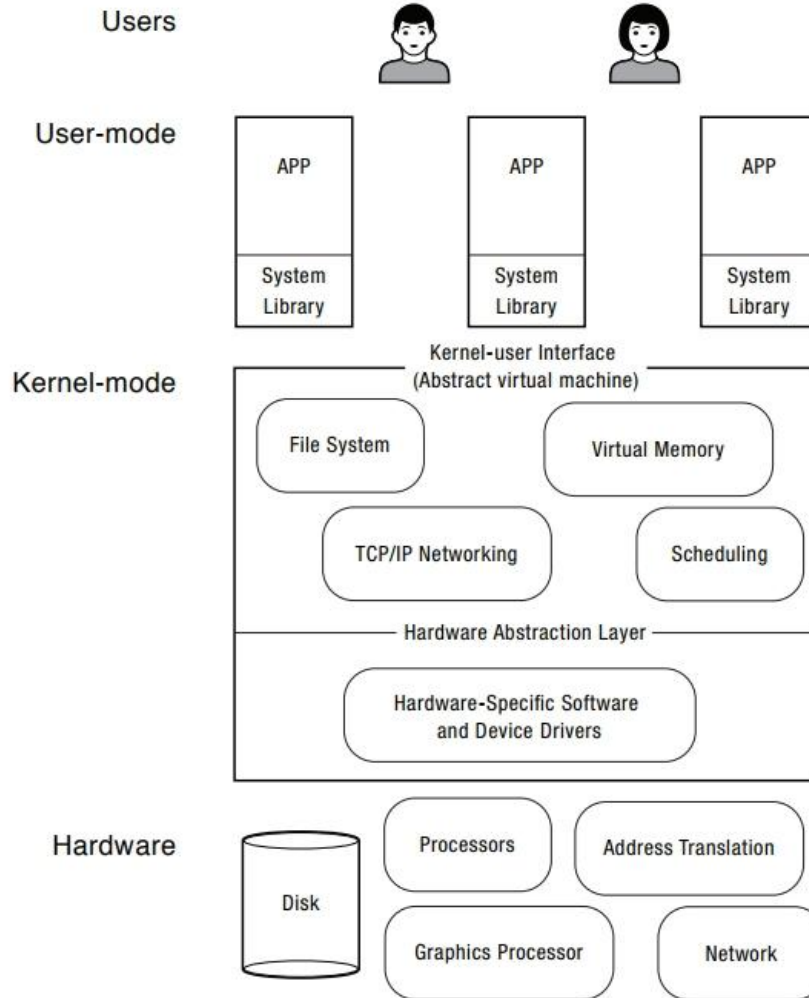
Usar herramientas en lugar de ayuda no especializada para aligerar una tarea de programación, incluso si tienes que desviarte para construir las herramientas y esperar tirar algunas de ellas después de haberlas usado.

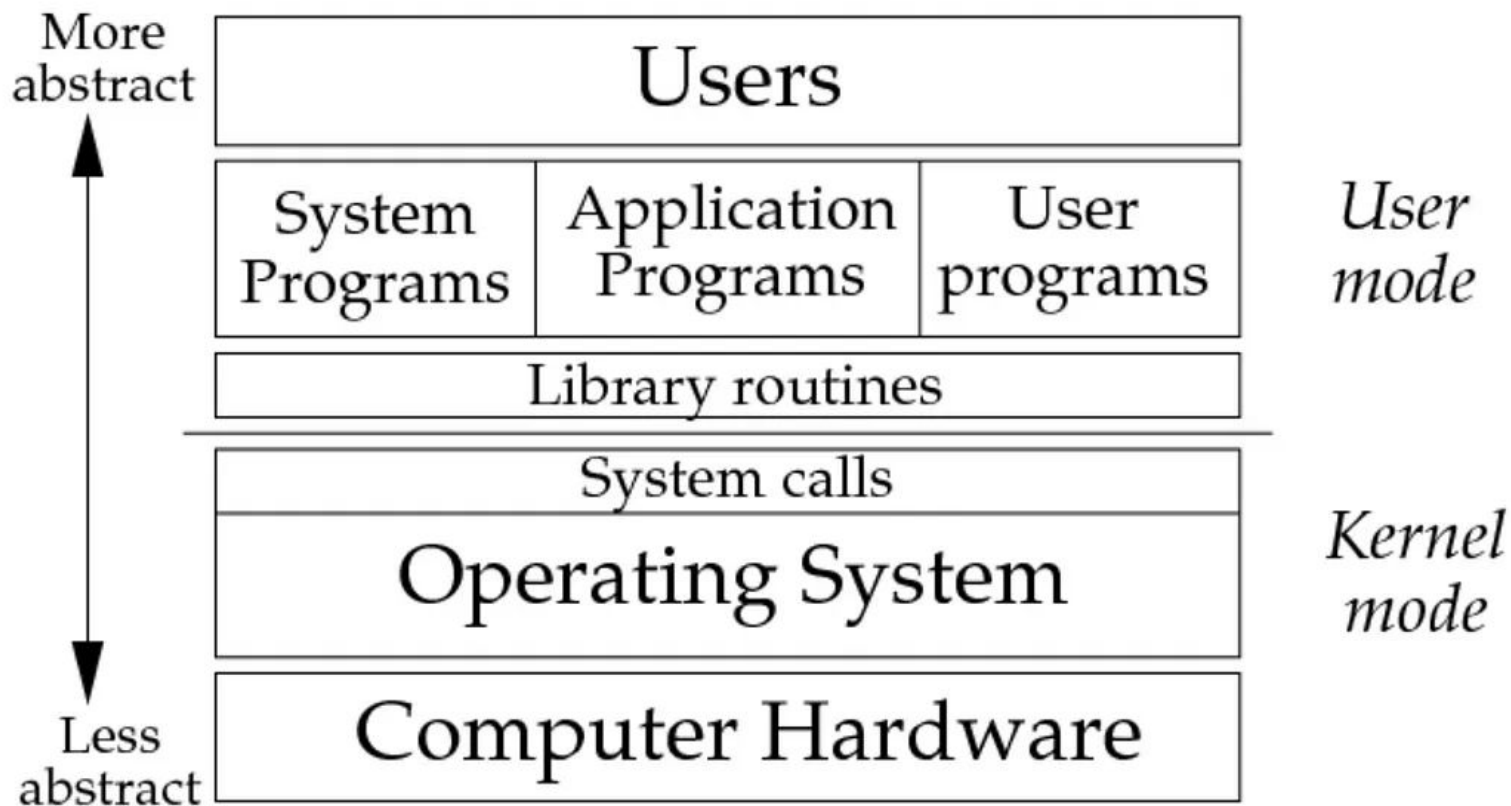




Sistema Operativo

Sistema Operativo





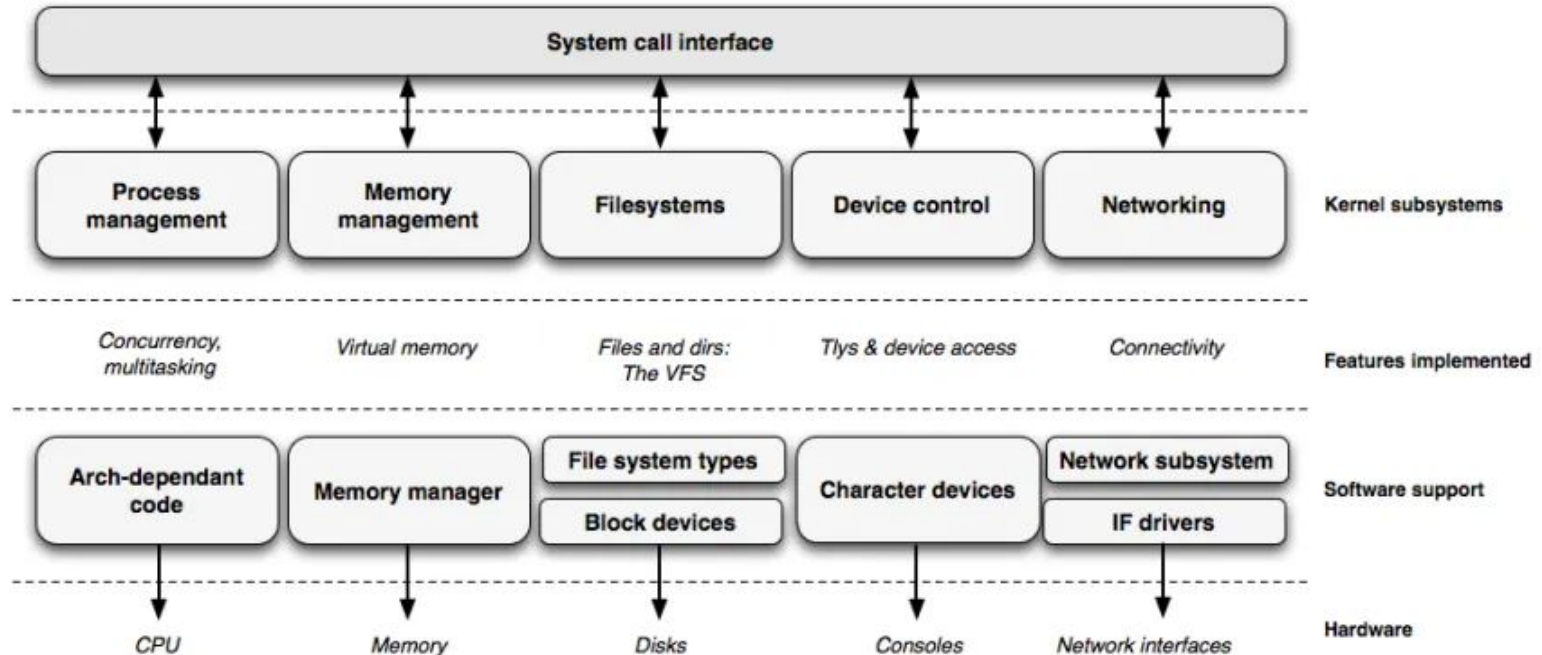


Linux: el Kernel

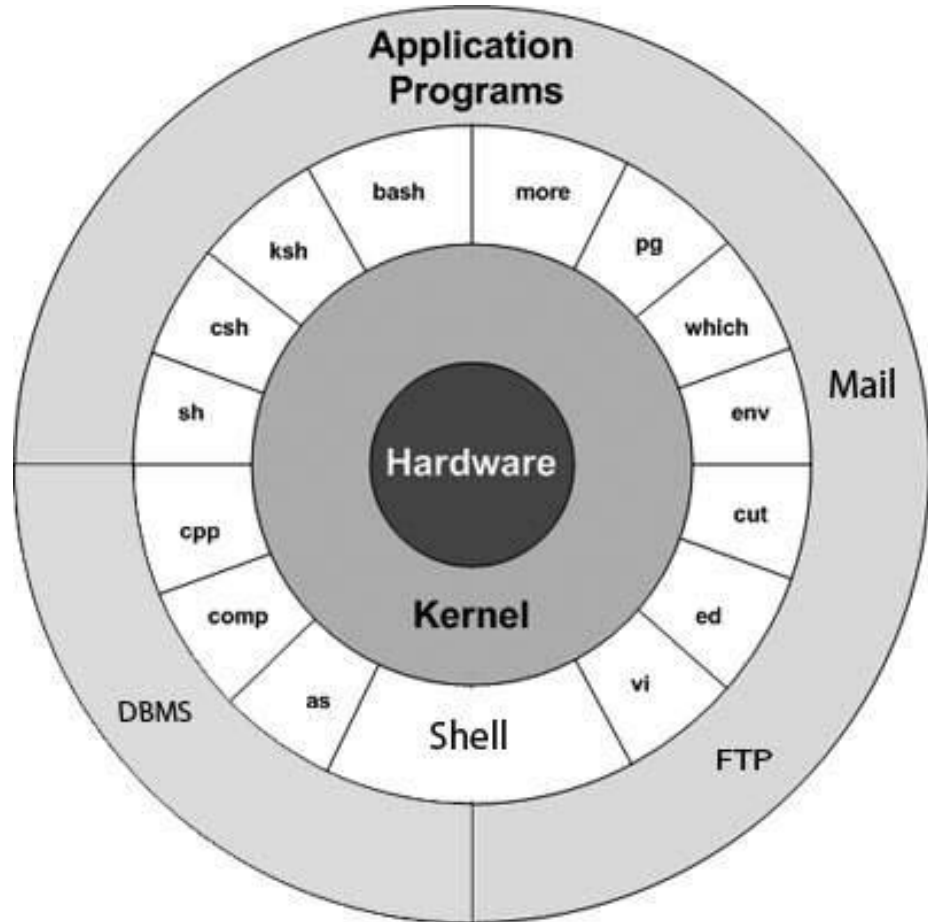
- La capa para la gestión de dispositivos específico
- y una **serie de servicios** para la gestión de dispositivos agnósticos del hardware que son utilizados por las aplicaciones.



Linux: el Kernel

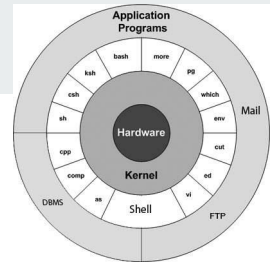
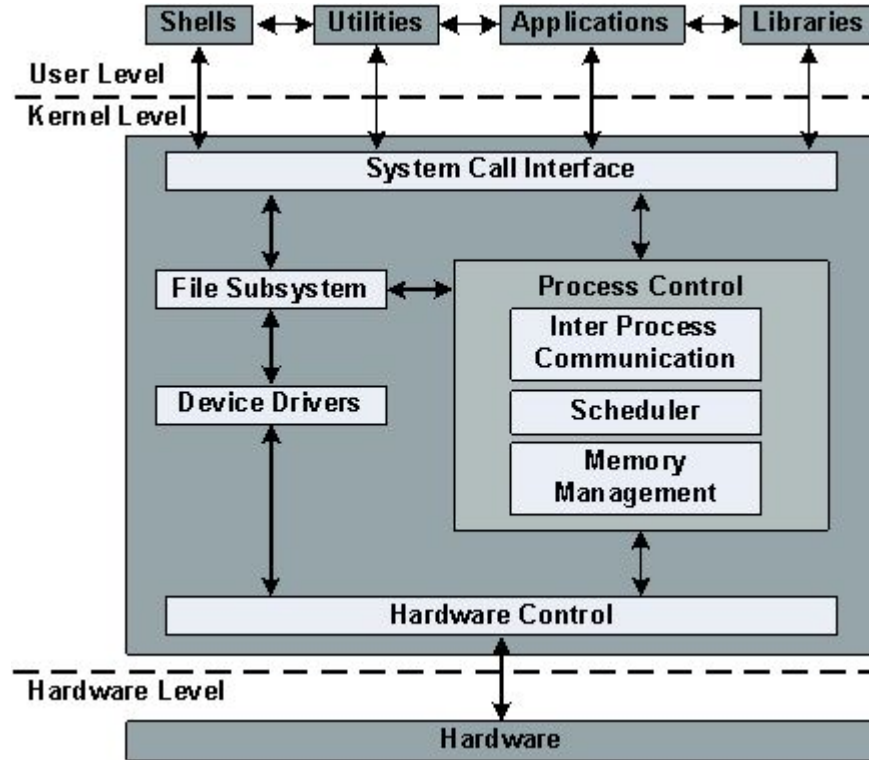


LINUX -> KERNEL



El Kernel

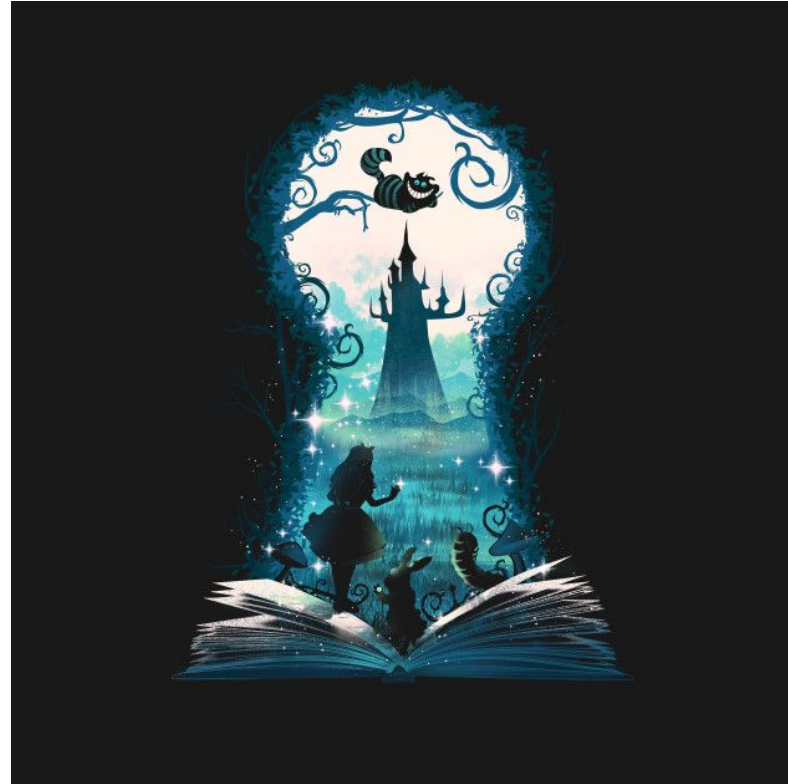
Kernel-land



¿Y qué pasa en User Land?

User-land: espacio donde viven las aplicaciones de usuario, a estas se las denominan **procesos**.

Cada proceso o programa en ejecución posee memoria con las instrucciones, los datos, el stack y el heap.

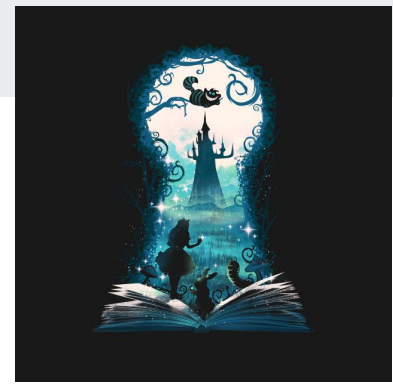


¿Y qué pasa en User Land?

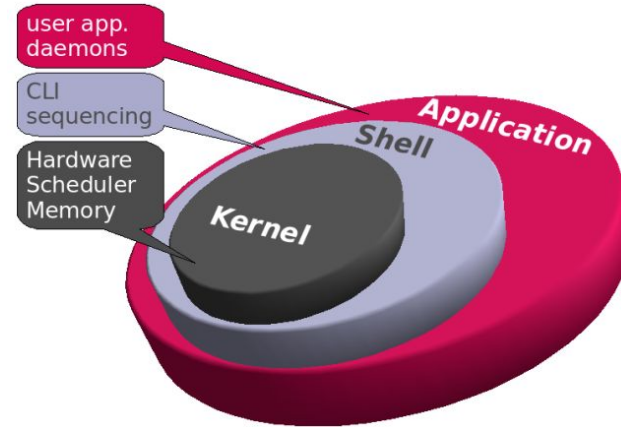
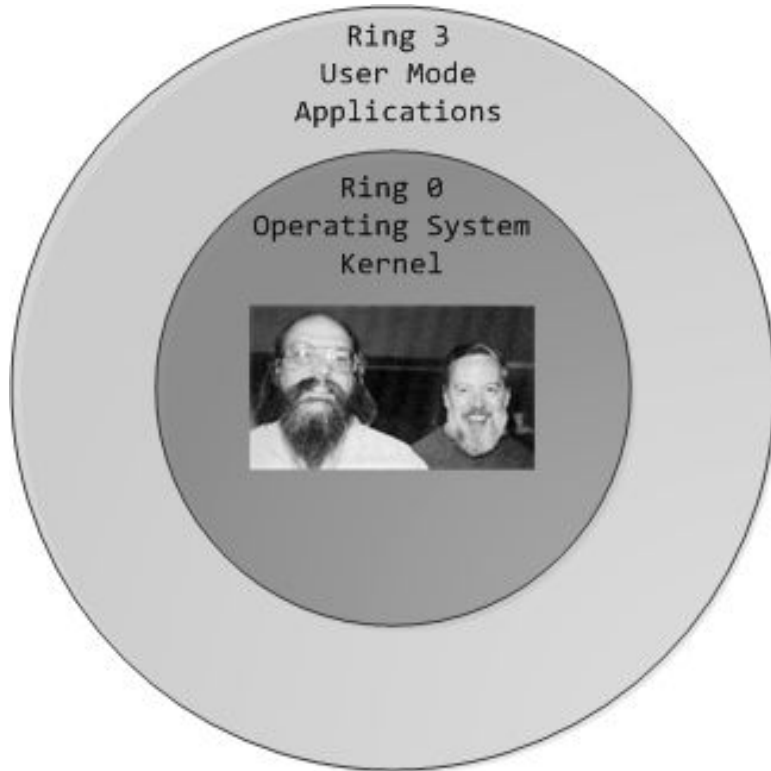


Los programas se están ejecutando!

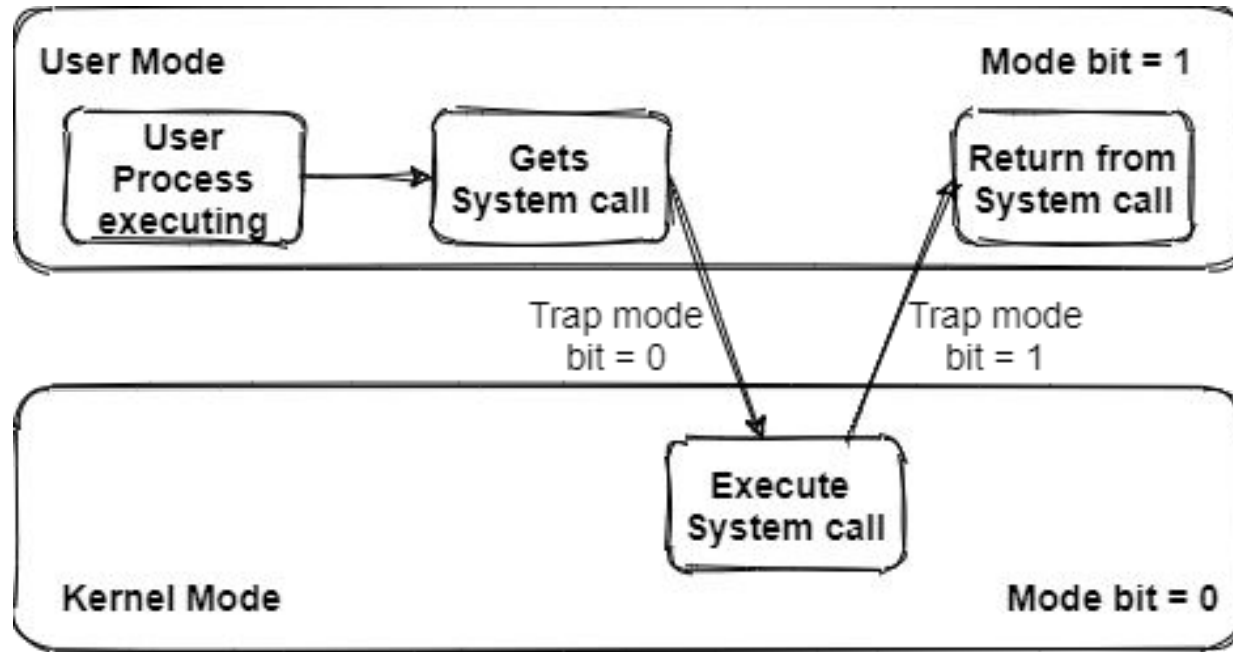
Las aplicaciones se ejecutan en un **contexto aislado, protegido y restringido** y mediante el uso de funciones que se encuentran en **bibliotecas** pueden utilizar los servicios de acceso al hardware o **recursos que el kernel proporciona**. El contexto de ejecución de las aplicaciones se denomina User Mode o modo usuario, más restrictivo, aislado y controlado.



User-land vs Kernel land



User-land vs Kernel land



Manual de linux



El manual de linux consta de 8 secciones, y puede leerse al invocar el comando `man` desde una terminal.

Sección 1: Programas disponibles para el usuario.

Sección 2: Rutinas del sistema Unix y C

Sección 3: Rutinas de bibliotecas del lenguaje C

Sección 4: Archivos especiales (dispositivos /dev)

Sección 5: Convenciones y formatos de archivos.

Sección 6: Juegos

Sección 7: Diversos (macros textuales, entre otras)

Sección 8: Procedimientos administrativos (daemons, etc)

Manual de linux



El manual se utiliza:

`$man wait`

`$man 2 wait`

(muestra la entrada en la sección 2 del manual)

`$ man man` muestra la entrada del comando man

System Calls



Una **system call** (llamada al sistema) es un punto de **entrada controlado al kernel**, permitiendo a un proceso solicitar que el kernel realice alguna operación en su nombre [KER](cap. 3).

El kernel expone una gran cantidad de servicios accesibles por un programa vía el **application programming interface (API)** de system calls.

System Calls



Algunas características generales de las system calls son:

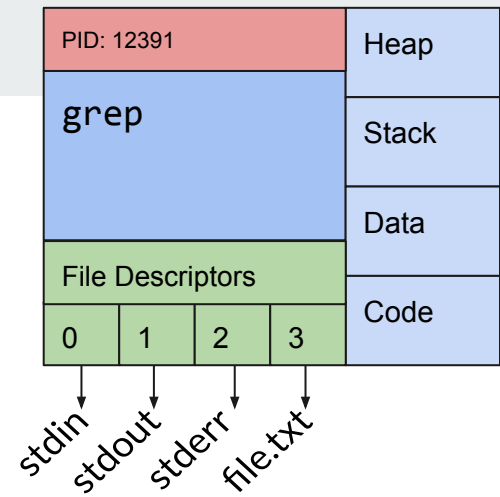
Una *system call* **cambia el modo del procesador de user mode a kernel mode**, por ende la CPU podrá acceder al área protegida del kernel.

El **conjunto de system calls es fijo**. Cada system call está identificada por un único número, que por supuesto no es visible al programa, éste sólo conoce su nombre.

Cada system call debe tener un conjunto de parámetros que especifican información que debe ser transferida desde el user space al kernel space.

Procesos

- Un proceso es un programa en ejecución
- Es algo dinámico
- Tienen una interna estructura propia
- Todos los procesos menos el kernel viven en user-land



Procesos

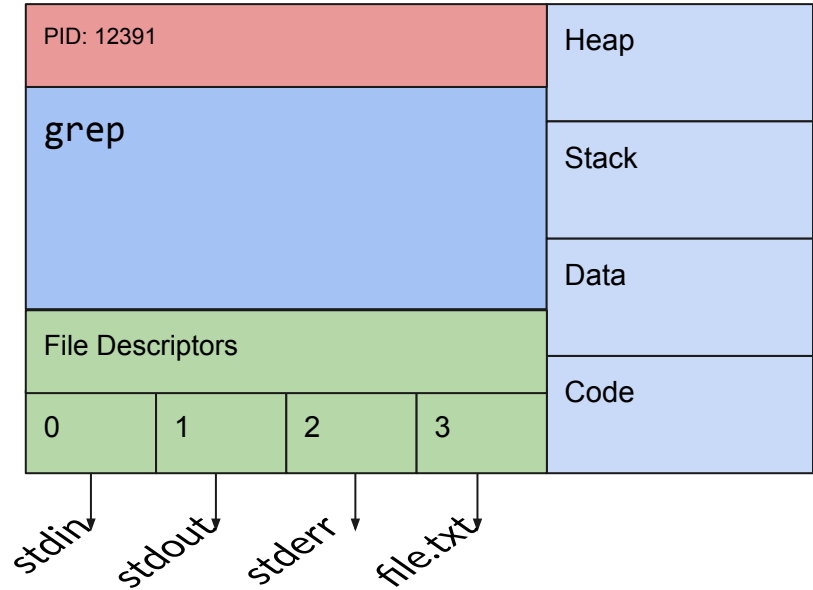
Partes básicas de un proceso:

PID:Process Id

Nombre del Programa

File Descriptors

Memoria:codigo,datos,stack,heap



El **kernel** posee una tabla llamada **Process Tables**, donde se guarda información de cada proceso, cuya entrada es el **pid del proceso**.

Procesos: File Descriptors



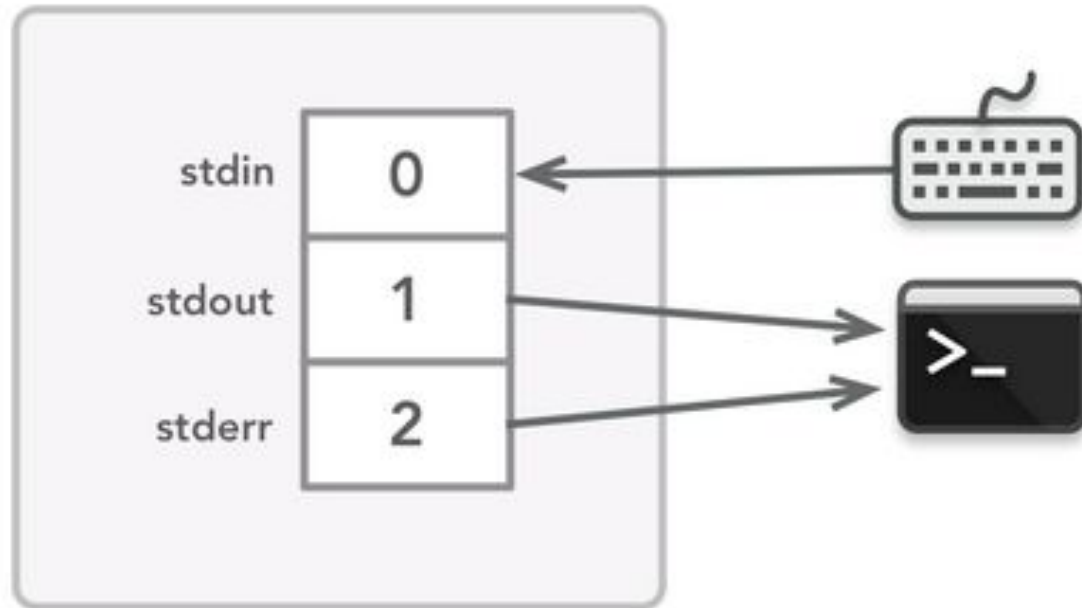
Los procesos tiene asociados los archivos abiertos que están usando. Estos archivos se identifican por un **número** denominado **file descriptor**, un número entero positivo.

Estos file descriptors se almacenan en una tabla en el kernel llamada **File Descriptor Table**. Hay una por proceso.

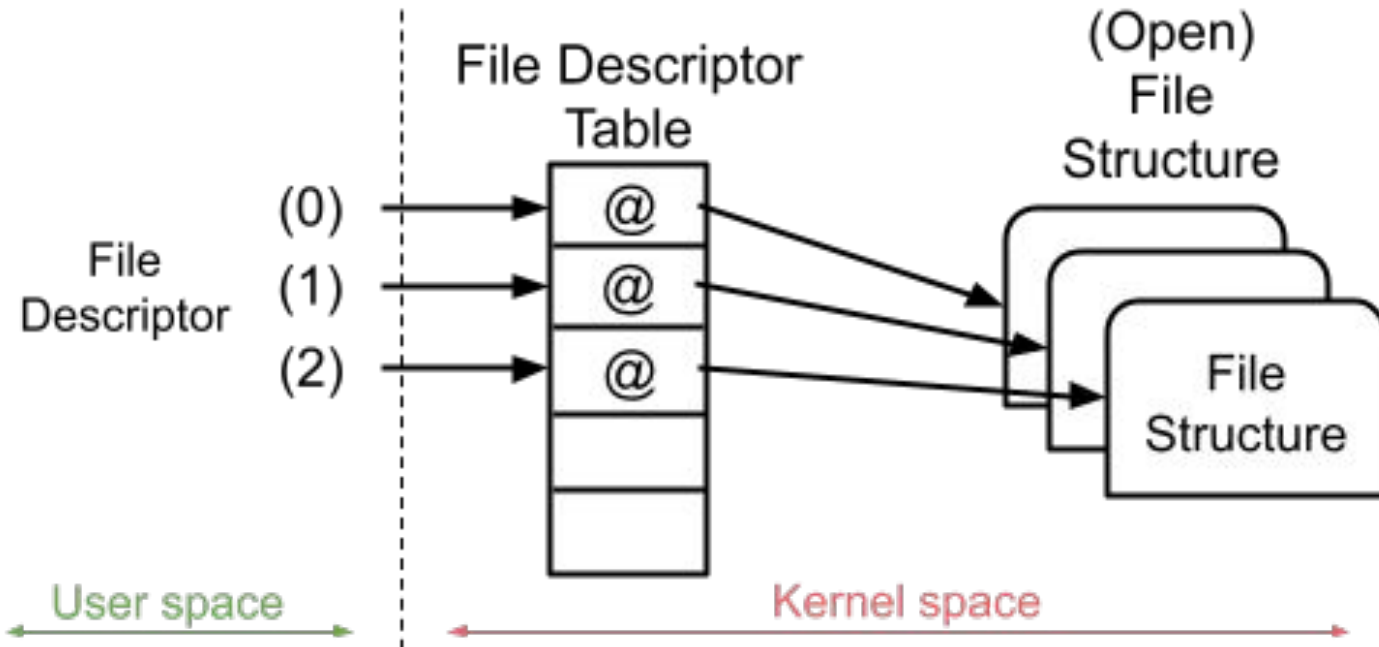
Existen 3 file descriptors que son creados cuando se crea un proceso:

fd=0	llamado Stdin de solo lectura	
fd=1	llamado Stdout	de solo escritura
fd=2	llamado StdErr	de solo escritura

Procesos: File Descriptors



Procesos: File Descriptors



El API resumida



fork(): Crea un proceso y devuelve su id.

wait(): Espera por un proceso hijo.

getpid(): Devuelve el pid del proceso actual.

exec(filename, argv): Carga un archivo y lo ejecuta.

exit(): Termina el proceso actual.

kill(pid): Termina el proceso cuyo pid es el parámetro.

pipe(): abre un buffer de memoria en el cual el proceso puede leer por un extremo y escribir por el otro.

dup:

System Call fork()



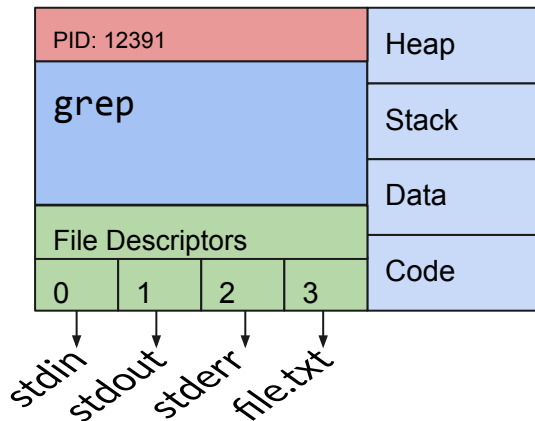
```
#include <unistd.h>  
pid_t fork(void);
```

La única forma de que un usuario cree un proceso en el sistema operativo UNIX es llamando a la system call fork.

El proceso que invoca a fork() es llamado proceso padre, el nuevo proceso creado es llamado hijo.

El nuevo proceso es una **copia exacta** de proceso padre, cuya única diferencia es su pid.

System Call fork()

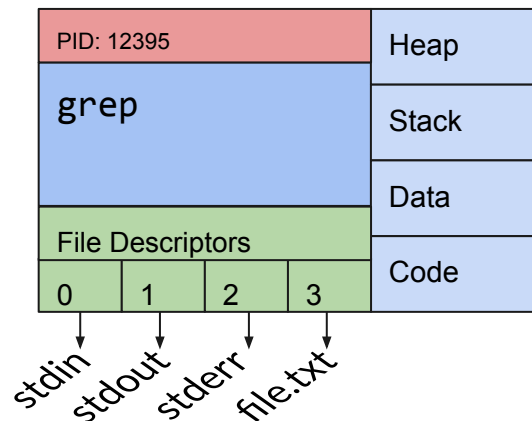
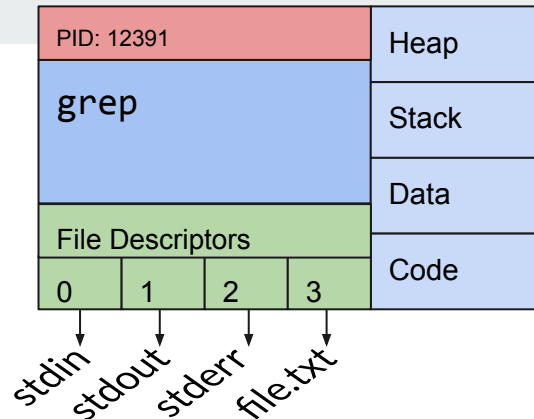


pid:12391

fork()

Pid:12391
(Padre)

Pid:12395
(Hijo)



System Call fork()



Notas:

- 1- Padre e hijo son copias exactas.
- 2- despues de fork() ambos se ejecutan por separado.
- 3- la única forma de saber quien es quien es mediante su **pid**.
- 4- el orden de ejecución de los procesos después del fork() no puede saberse.

System Call fork()



```
int pid = fork();
if(pid == 0){
    printf("child: exiting\n");
} else if(pid > 0){
    printf("parent: child=%d\n", pid);
} else {
    printf("fork error\n");
}
```

System Call fork()

```
int pid = fork();  
if(pid == 0){  
    printf("child: exiting\n");  
} else if(pid > 0){  
    printf("parent: child=%d\n", pid);  
} else {  
    printf("fork error\n");  
}
```

Solo lo ejecuta el hijo



Solo lo ejecuta el padre



Creación de un Proceso:¿Que hace fork?:



- Crea y asigna una nueva entrada en la **Process Table** para el nuevo proceso.
- Asigna un número de ID único al proceso hijo (**pid**).
- Crea una copia del proceso padre.
- Realiza ciertas operaciones de I/O, abre stdin, stdout, stderr.
- Devuelve el número de ID del hijo al proceso padre, y un 0 al proceso hijo



System Call wait()

Esta system call se utiliza para **esperar un cambio de estado en un proceso hijo** del proceso que realiza la llamada, y obtener información sobre el proceso hijo cuyo estado ha cambiado.

Se considera que un cambio de estado es:

- El hijo termina su ejecución
- El fue parado tras recibir un signal
- El hijo continúa su ejecución tras haber recibido un signal



System Call wait()

```
#include <sys/wait.h>  
pid_t wait(int *_Nullable wstatus);
```

wait() retorna el pid del proceso que sufrió el cambio de estado. Y copia el estado de salida del proceso en cuestión en la dirección wstatus.

Si no se desea info del estado se le pasa la direccion 0.



System Call wait()

```
#include <sys/wait.h>
pid_t wait(int *_Nullable wstatus);
```

wait() retorna el pid del proceso que sufrió el cambio de estado. Y copia el estado de salida del proceso en cuestión en la dirección wstatus.

Si no se desea info del estado se le pasa la dirección 0.

Esta llamada es bloqueante



System Call wait()

```
int pid = fork();
if(pid == 0){
    printf("child: exiting\n");
    exit(0);
} elseif(pid > 0){
    printf("parent: child=%d\n", pid);
    pid = wait((int *) 0);
    printf("child %d is done\n", pid);
} else {
    printf("fork error\n");
}
```

System Call getpid()



```
#include <unistd.h>  
pid_t getpid(void);  
pid_t getppid(void);
```

Getpid(): devuelve el process id del proceso llamador.

getppid(): devuelve el process id del padre del proceso llamador.

System Call exit()



Generalmente un proceso tiene dos formas de terminar:

La anormal: a través de recibir una señal cuya acción por defecto es terminar el programa.

La normal: a través de invocar a la system call exit()

System Call `execve()`



Si únicamente tuviéramos las system calls `fork()` y `wait()`. Se podrían crear procesos que siempre son una copia exacta del padre sería un poco engorroso.

... y qué tal si pudiera cambiar el contenido del hijo...

System Call `execve()`



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

```
int execve(const char *pathname, char *const _Nullable argv[],char  
*const _Nullable envp[]);
```

`execve()` ejecuta el programa al que hace referencia el nombre de `pathname`, con los argumentos que se le envían por `argv[]`.


Esto hace que el programa que está ejecutando actualmente por el proceso llamador **sea reemplazado por un nuevo programa**, con una pila, un montón y segmentos de datos (inicializados y no inicializados) recién inicializados.

System Call `execve()`

Ojo que no se crea ningún proceso solo se reemplaza
Un programa en ejecución por otro.



System Call `execve()`



```
#include<stdio.h>
#include<unistd.h>
int main (){
    char *argv[3];
    argv[0] = "echo";
    argv[1] = "hello";
    argv[2] = 0;

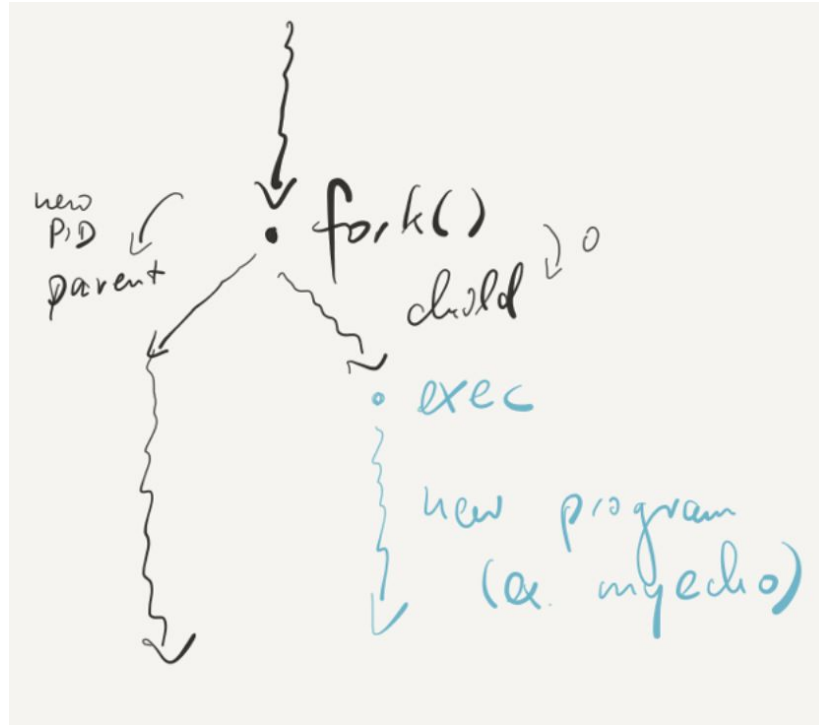
    printf("este es un proceso cuyo pid es: %i \n",getpid());
    printf("Ahora lo vamos a pisar y hacer que el mismo proceso\n");
    printf("ejecute un programa perdiendo todo y sustituyendo \n");
    printf("todo por el nuevo programa que se iniciara a ejecutar \n");

    execve("/bin/echo", argv, NULL);

    printf("exec error\n");
    printf("esto nunca se ejecuta\n");
}
```

System Call `execve()`

Y si lo combinamos con `fork()`...



System Call `execve()`

```
int main (){
    char *argv[3];
    int pid = fork();
    if(pid == 0){
        argv[0] = "echo";
        argv[1] = "hello yo soy el comando echo!!!";
        argv[2] = 0;

        execve("/bin/echo", argv, NULL);
        printf("esto no debe ejecutarse\n");
    } else if(pid > 0){

        printf("parent: child=%d\n", pid);
        pid = wait((int *) 0);
        printf("child %d is done\n", pid);

    } else {
        printf("fork error\n");
    }
}
```

System Call dup()



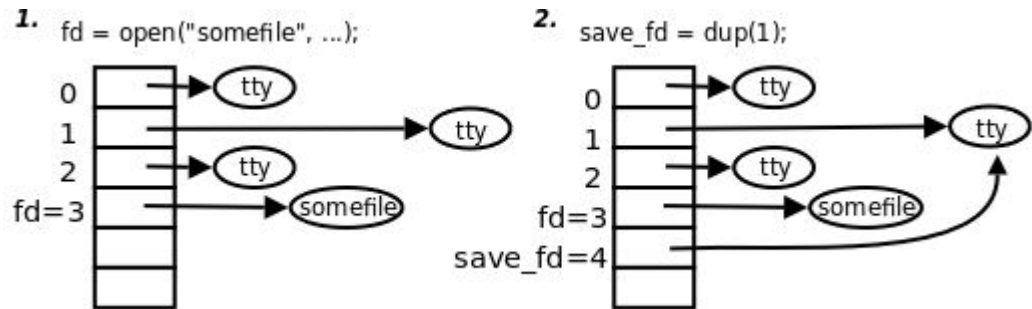
```
#include <unistd.h>  
int dup(int fildes);  
int dup2(int fildes, int fildes2);
```

Estas System Calls duplican un file descriptor, el cual puede ser utilizado indiferentemente entre el y el duplicado.

System Call dup()

```
#include <unistd.h>
int dup(int fildes);
```

dup() retorna un nuevo file descriptor que es copia del enviado como parámetro, obteniendo el primer file descriptor libre que se encuentre en la **File Descriptor Table**.



System Call dup()

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

```
int main (){

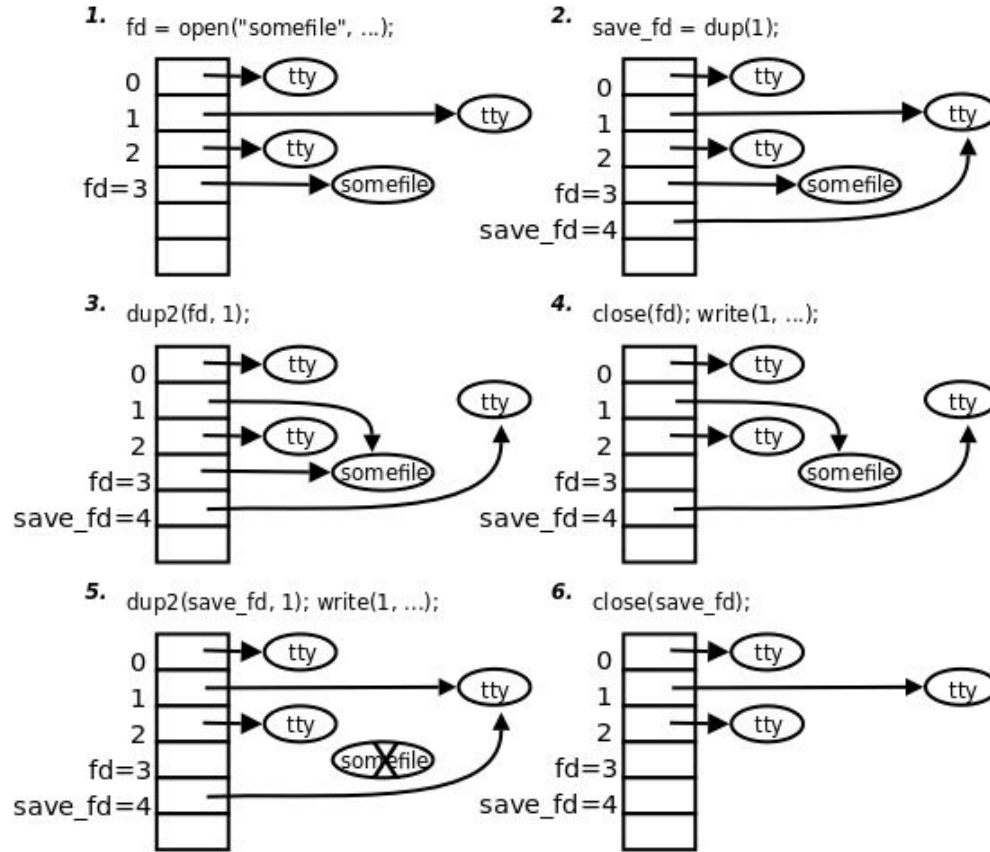
    int dup_fd;
    char msg1[]="Se escribe en el stdout\n";
    char msg3[]="se escribe desde el file descriptor duplicado\n";

    dup_fd=dup(1);
    printf("dup_fd: %i\n",dup_fd);
    write(1,msg1,sizeof(msg1)-1);
    write(dup_fd,msg3,sizeof(msg3)-1);
    close(dup_fd);

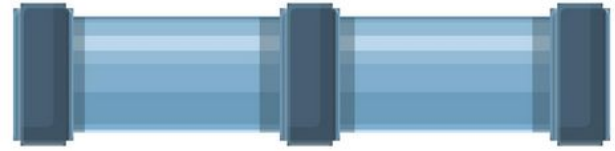
    write (1, msg1, sizeof(msg1)-1);
}
```

System Call dup()

G



System Call pipe()

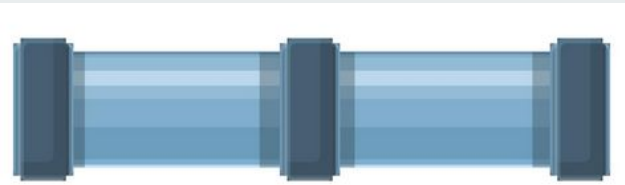


```
#include <unistd.h>  
int pipe(int pipefd[2]);
```

Un pipe es un **pequeño buffer en el kernel** expuesto a los procesos como un par de files descriptors, uno para escritura y otro para lectura. Al escribir datos en el extremo el pipe hace que estos estén disponibles en el otro extremo para ser leídos.

ES UN CANAL UNIDIRECCIONAL!!!

System Call pipe()



Int pipefd[2]

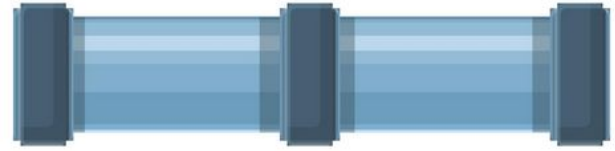


escribo
pipefd[1]



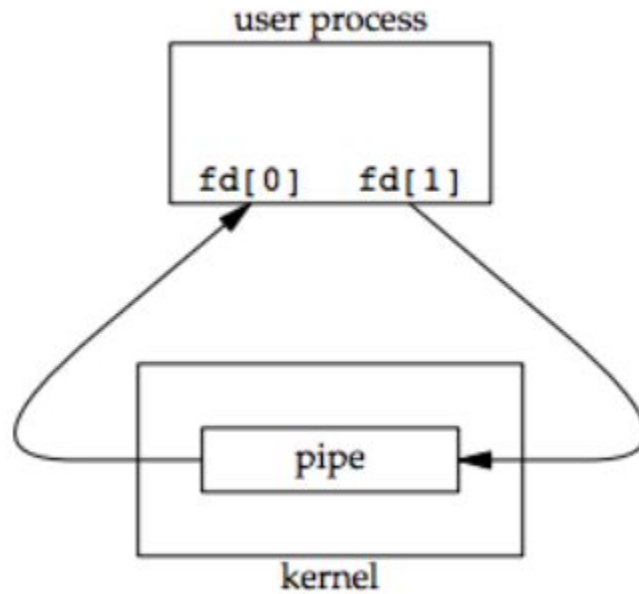
leo
pipefd[0]

System Call pipe()

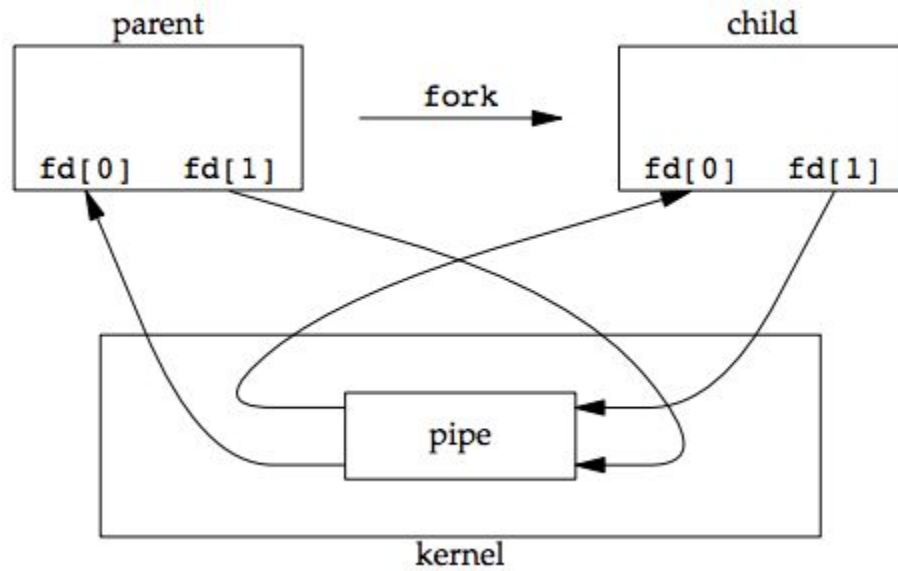


Puedo hacer que dos procesos compartan informacion.

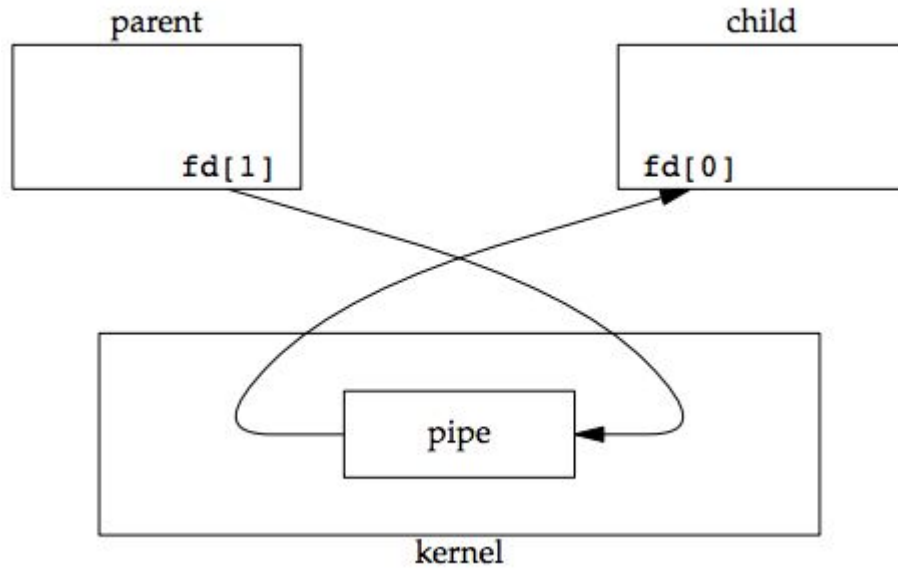
System Call pipe()



System Call pipe()



System Call pipe()



System Call pipe()



```
int p[2];
char *argv[2];
argv[0] = "wc";
argv[1] = 0;
pipe(p);
if(fork() == 0) {
    close(0);
    dup(p[0]);
    close(p[0]);
    close(p[1]);
    exec("/bin/wc", argv);
} else {
    close(p[0]);
    write(p[1], "hello world\n", 12);
    close(p[1]);
}
```