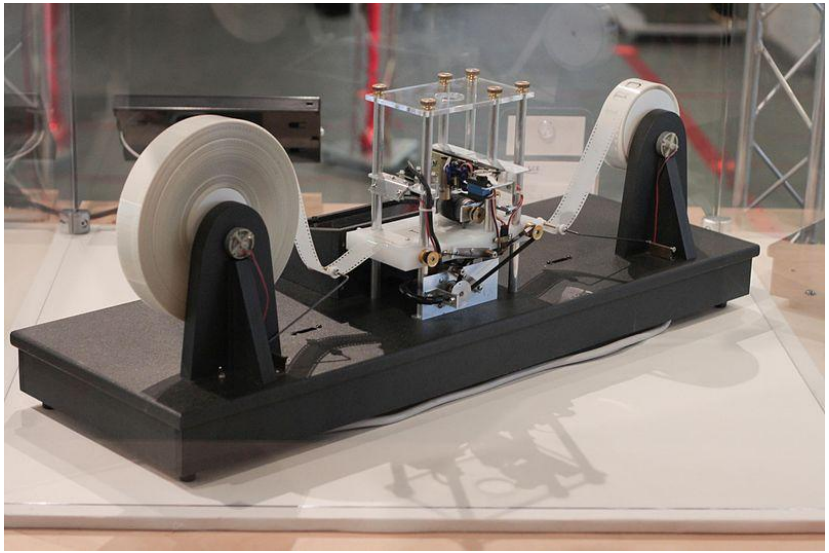
Sistemas Operativos
Fisop 2024

Introducción

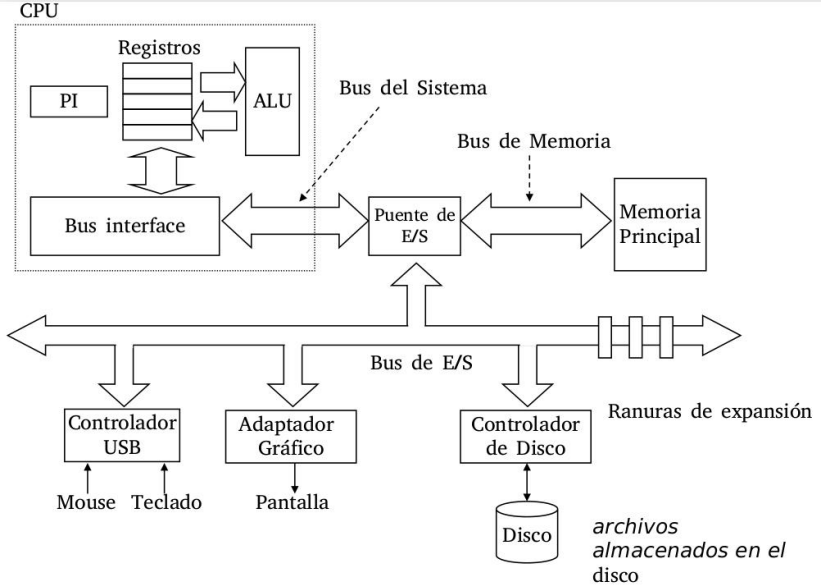
¿Qué es un Sistema Operativo?

¿Si te digo sistema operativo qué es en lo primero que pensas?

Alan Turing



John Von Neuman



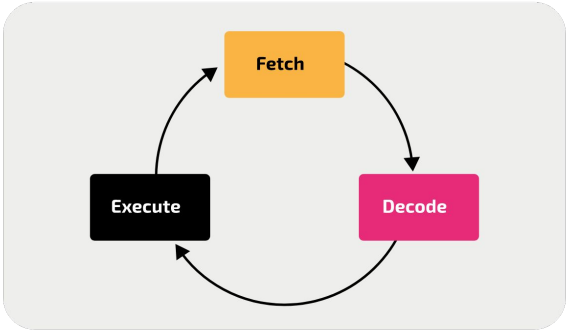
¿Qué pasa cuando un programa se ejecuta?

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

La decodifica (**decode**)

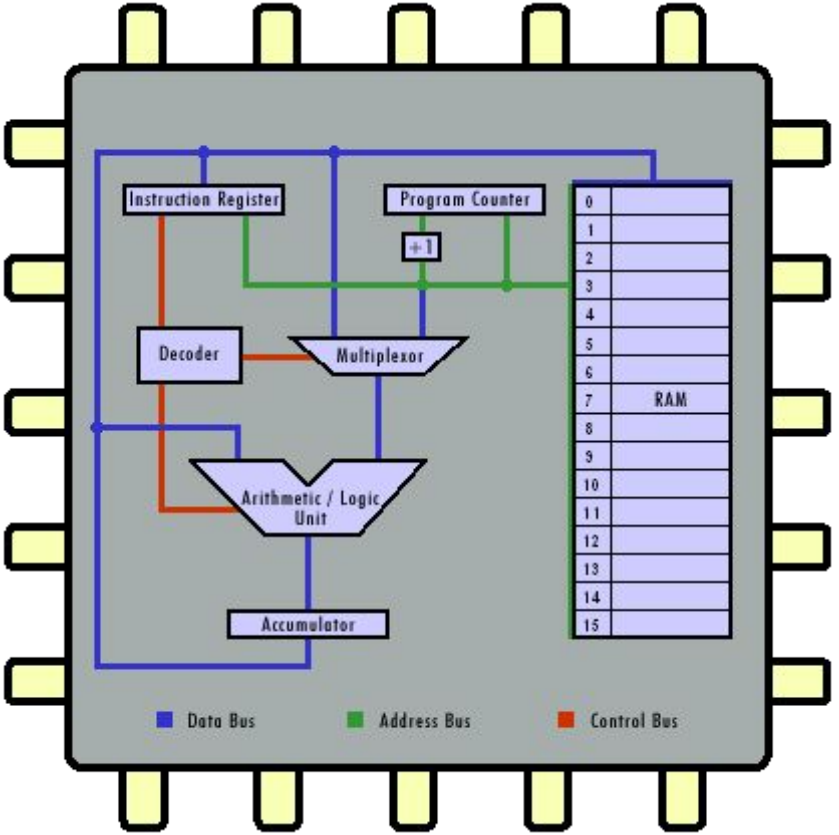
Finalmente la ejecuta (**execute**)

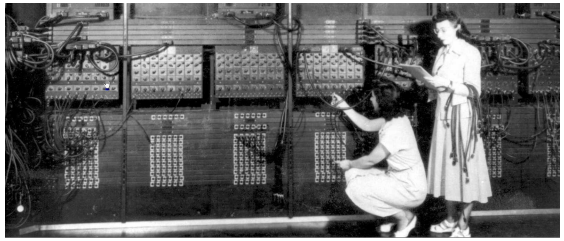
fetch-decode-execute



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

Primera computadora de Latinoamérica

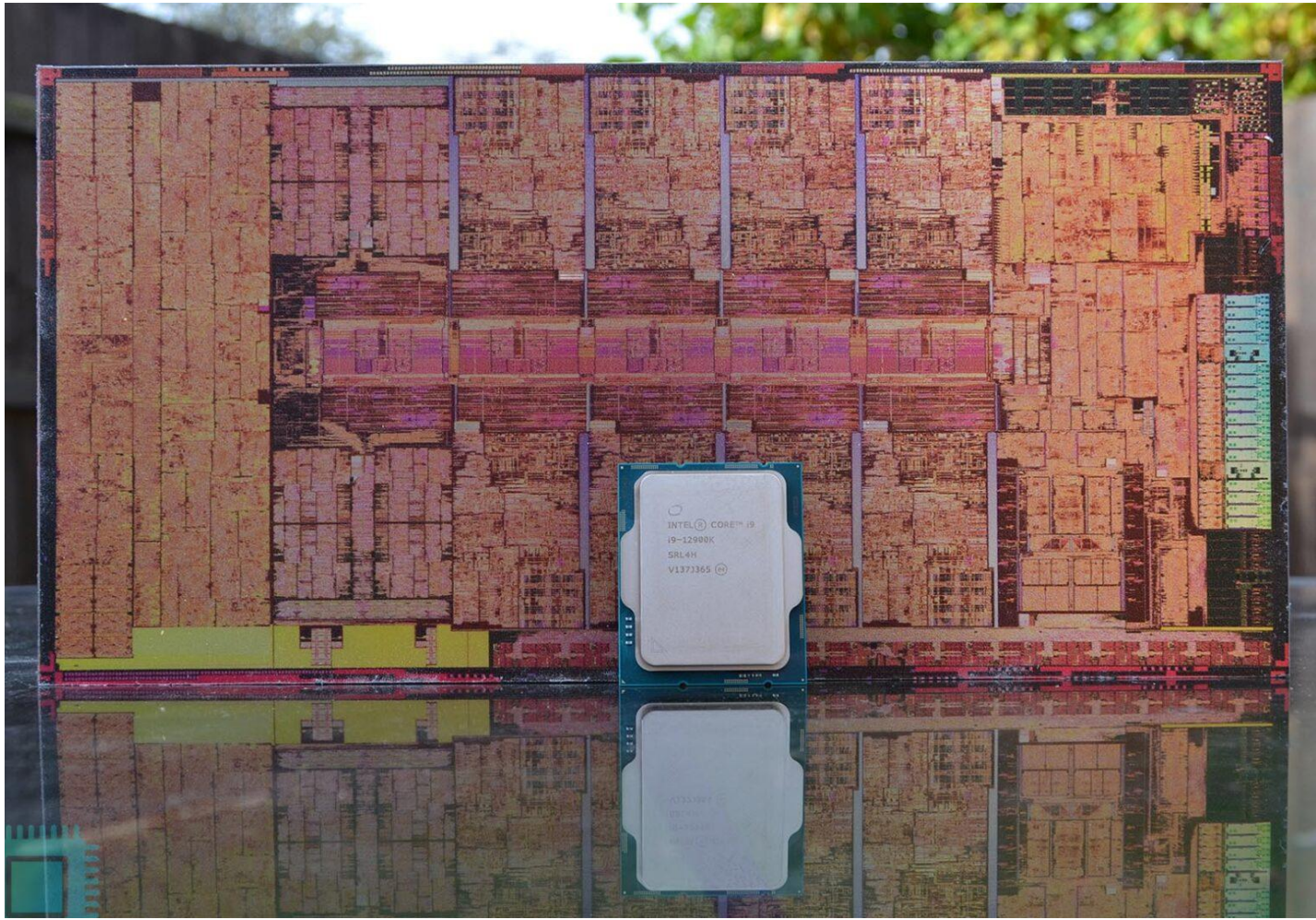
Clementina (Ferranti Mercury) - 1960 - Pabellón I Ciudad Universitaria

Ver mas en:

- *https://www.dc.uba.ar/la-serie-clementina/*
- *Historia de la informática en Latinoamérica y el Caribe: investigaciones y testimonios - Marcelo Savio Carvalho - 2009*

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."

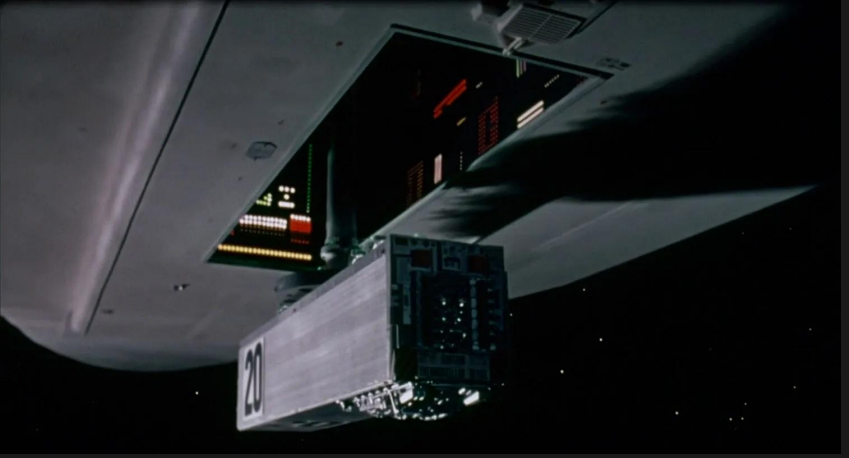


Motivación

Rol del sistema operativo moderno

Conceptos clave:

- Definiciones fundamentales
- Funciones del SO
- Virtualización



Bomba #20: En otras palabras, todo lo que sé realmente sobre el mundo exterior me lo transmiten a través de mis conexiones eléctricas.

Doolittle: ¡Exactamente!

Bomba #20: ¿Por qué? Eso significaría que... en realidad no sé con certeza cómo es el universo exterior.

Doolittle: ¡Eso es! ¡Eso es!

Bomba #20: Intrigante. Ojalá tuviera más tiempo para hablar de este asunto.

Doolittle: ¿Por qué no tienes más tiempo?

Bomba #20: Porque debo detonar en 75 segundos.

Dark Star (1974) de John Carpenter

¿Qué Hace un Sistema Operativo?

El trabajo de un sistema operativo es **compartir** una computadora entre múltiples programas y **proporcionar** un **conjunto de servicios** más útil de lo que el hardware por sí solo soporta.

Un sistema operativo **gestiona** 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 Hace un Sistema Operativo?

Un sistema operativo comparte el hardware entre múltiples programas para que se ejecuten (o parezcan ejecutarse) al mismo tiempo.

Finalmente, los sistemas operativos proporcionan formas controladas para que los programas interactúen, de modo que puedan compartir datos o trabajar juntos.

¿Qué Hace un Sistema Operativo?

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

Diseñar una buena interfaz resulta ser difícil:

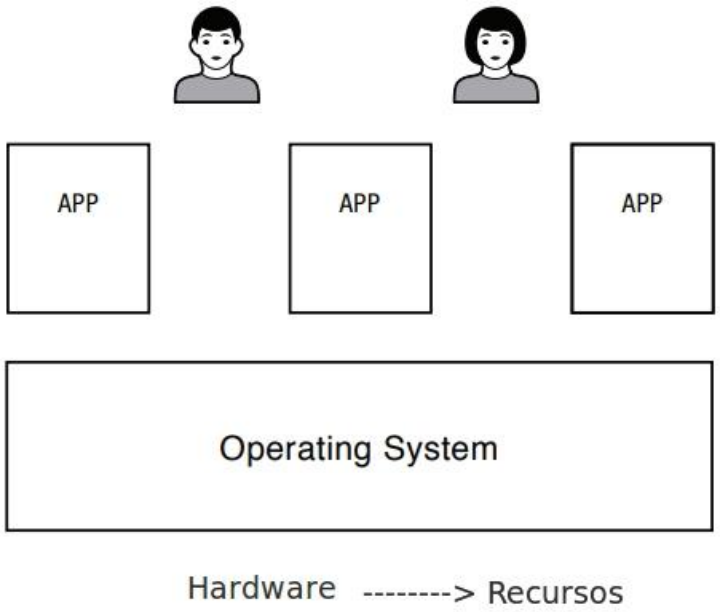
- Por un lado, nos gustaría que la interfaz fuera **simple y reducida** porque eso facilita hacer que la implementación sea correcta.
- Por otro lado, podríamos estar tentados a ofrecer muchas características sofisticadas a las aplicaciones. El truco para resolver esta tensión es diseñar interfaces que se basen en ***unos pocos mecanismos que se puedan combinar para proporcionar una gran generalidad***.

¿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 usuario** interactuan 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.

Funciones de un S.O.

Referee (Árbitro)

1. **Gestión de Recursos**:
    - Los sistemas operativos **gestionan los recursos compartidos** entre diferentes aplicaciones en una misma máquina.
    - Pueden detener un programa y iniciar otro según sea necesario.
    - Aíslan aplicaciones entre sí, evitando que errores en una afecten a las demás.
    - Protegen al sistema y a las aplicaciones de virus informáticos.

2. **Distribución de Recursos**:
    - Deciden qué aplicaciones reciben cuáles recursos y cuándo, ya que todas comparten los recursos físicos disponibles.

Funciones de un S.O.

Illusionist (Ilusionista)

1. **Abstracción del Hardware**:
    - Proveen una abstracción del hardware físico para simplificar el diseño de aplicaciones.
    - Ofrecen la ilusión de memoria casi infinita y uso exclusivo de procesadores, independientemente de la cantidad real de memoria física o número de procesadores disponibles.
    - Permiten escribir aplicaciones sin preocuparse por los detalles físicos del sistema.

Glue (Conector)

1. **Servicios Comunes**:
    - Facilitan el uso compartido de información entre aplicaciones, como copiar y pegar, y acceso a archivos.
    - Proveen rutinas de interfaz de usuario comunes para una apariencia y experiencia uniforme en el sistema.
    - Actúan como capa de separación entre las aplicaciones y los dispositivos de entrada/salida (I/O), permitiendo que las aplicaciones funcionen independientemente del hardware específico en uso.

Virtualización

La principal manera en que el SO logra esto es a través de una técnica general que llamamos virtualización. Es decir, el SO toma un recurso físico (como el procesador, la memoria o un disco) y lo transforma en una forma virtual más general, poderosa y fácil de usar. Por lo tanto, a veces nos referimos al sistema operativo como una máquina virtual.

Virtualización = Abstracción

Se visualiza:

- La CPU
- La Memoria
- El Tiempo (concurrencia)
- Los Periféricos

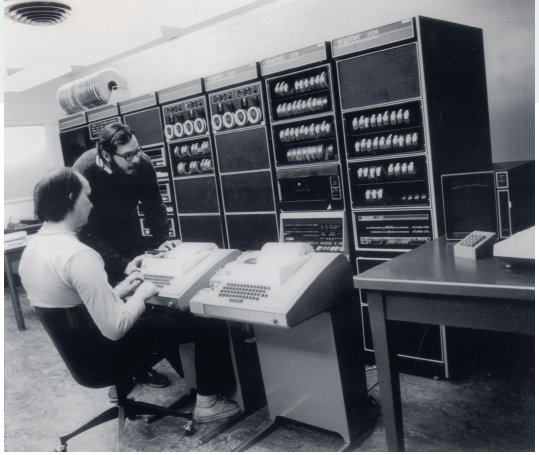
Unix

El caso de estudio fundamental

Conceptos clave:

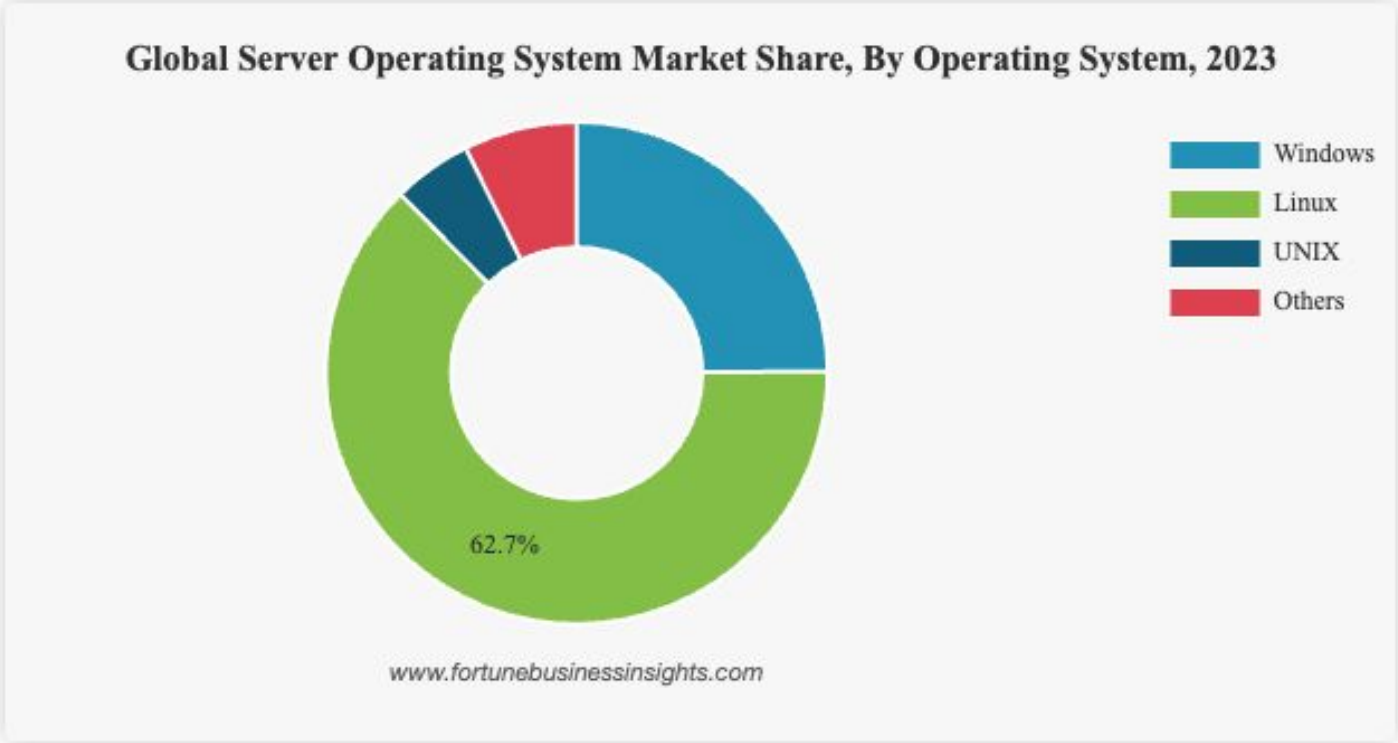
- Historia de Unix
- Decisiones de diseño de Unix

Unix



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

- El sistema operativo Unix de Ken Thompson y Dennis Ritchie proporciona una interfaz reducida cuyos mecanismos se combinan bien, ofreciendo un grado sorprendente de generalidad.



https://docs.aws.amazon.com/wellarchitected/latest/operational-readiness-reviews/wa-operational-readiness-reviews.html

Unix

Unix: Philosophy

Simplicidad en Funcionalidades:

- Cada programa debe realizar **una única tarea bien**. Para nuevas funcionalidades, crea nuevos programas en lugar de añadir complejidad a los existentes.

Interoperabilidad y Simplicidad en la Salida:

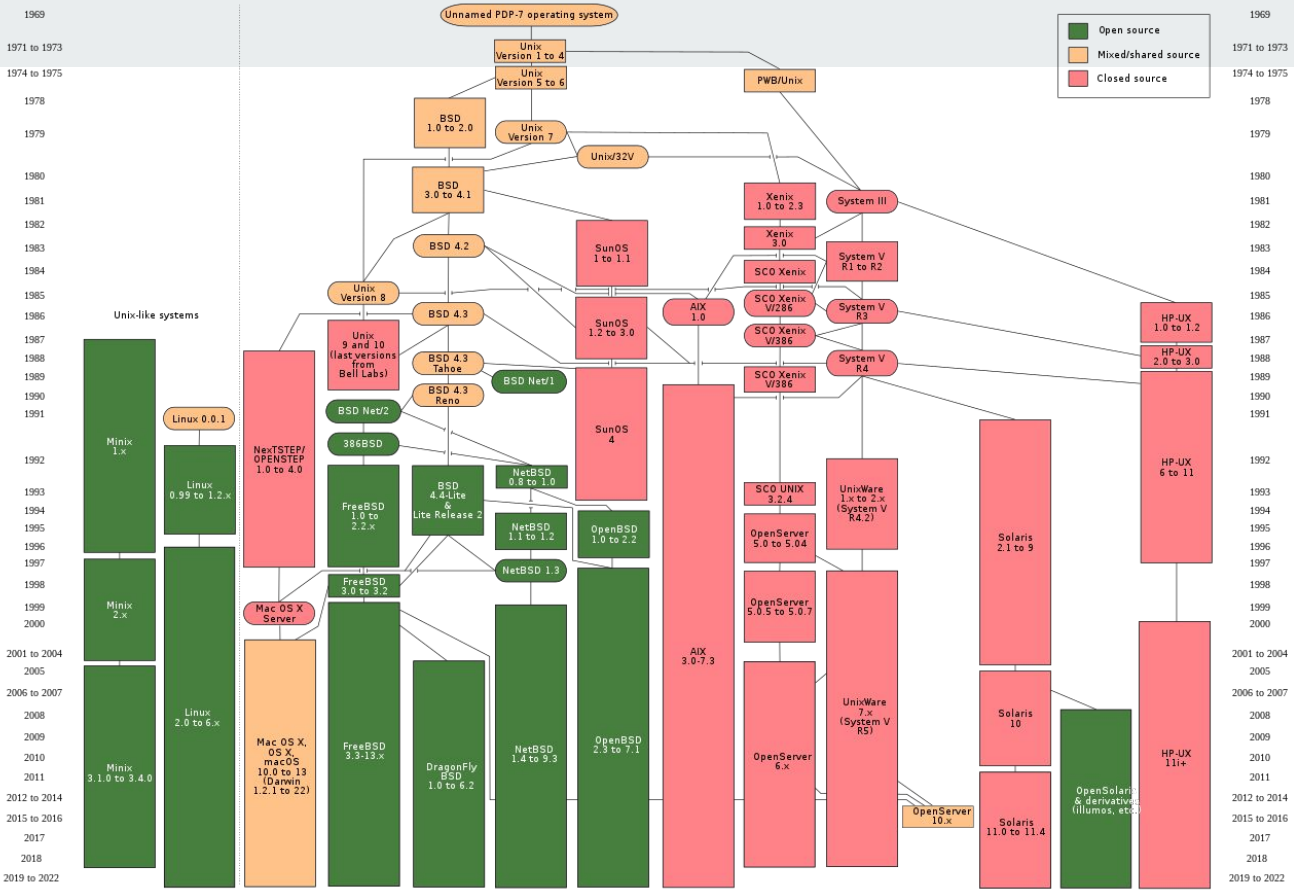
- Diseña la **salida de cada programa** para que pueda ser utilizada como **entrada de otros programas**. Evita formatos complejos o interactivos.

Pruebas Tempranas y Refactorización:

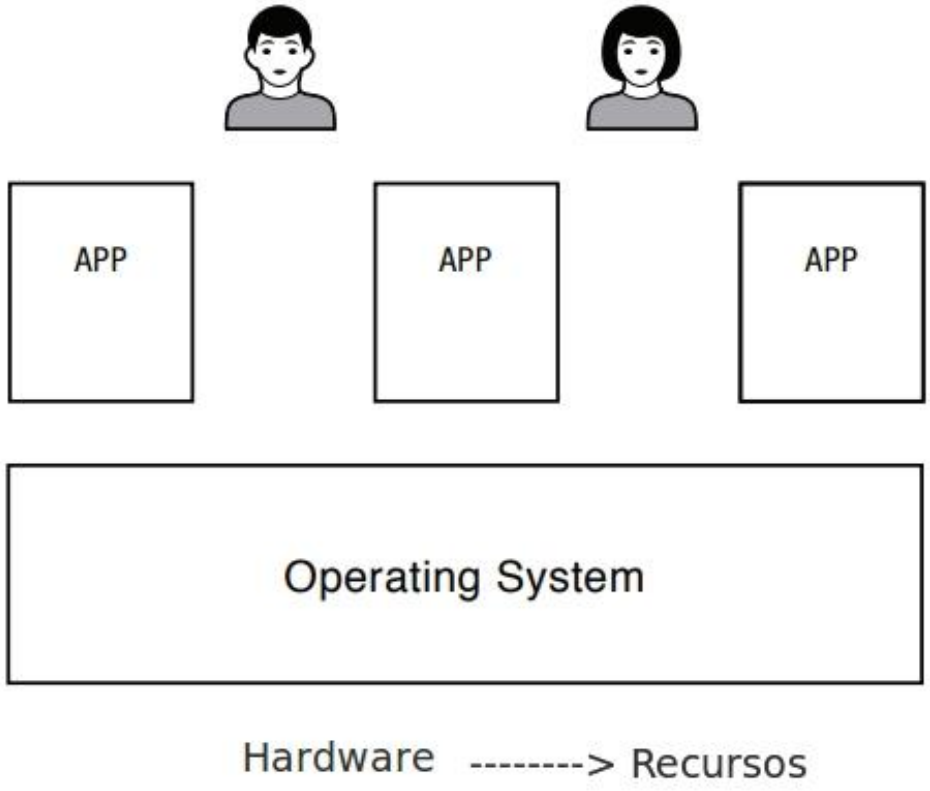
- Diseña software, incluidos sistemas operativos, para ser probado temprano, idealmente en semanas. No dudes en descartar y rehacer partes ineficaces.

Uso de Herramientas para Eficiencia:

- Utiliza herramientas especializadas para simplificar tareas de programación, incluso si implica construir herramientas temporales que se descarten después de su uso.

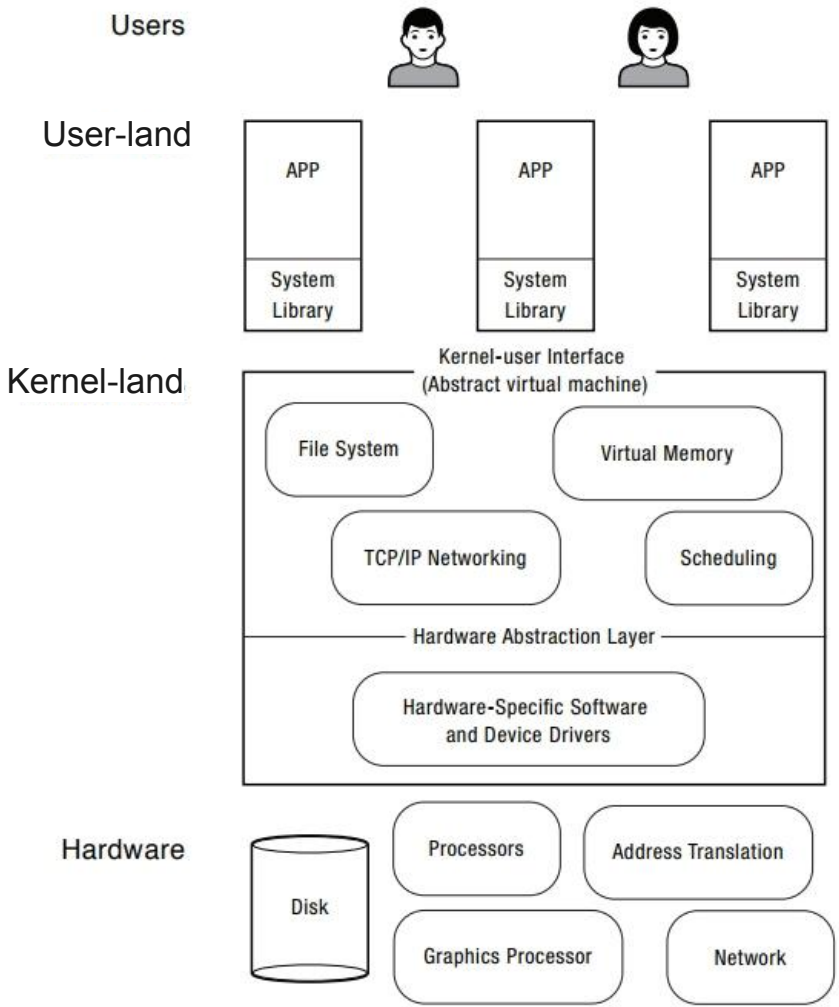


Unix Evolution



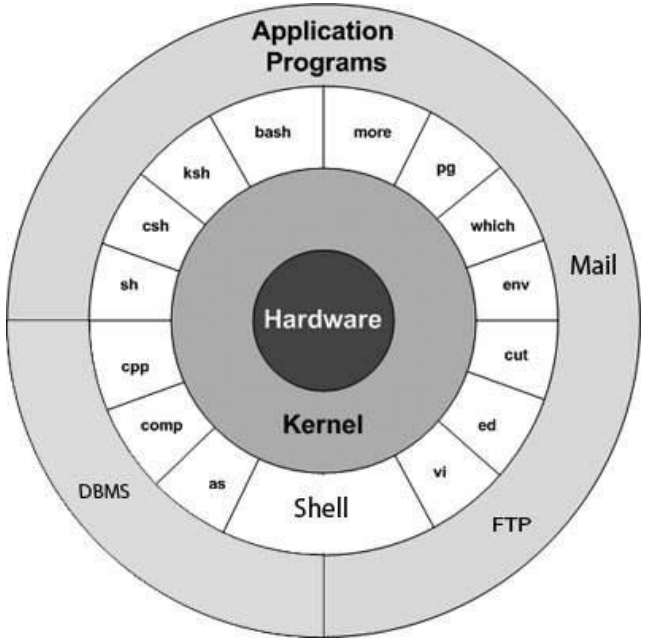
Sistema Operativo

Sistema Operativo Unix-like



Linux: el Kernel

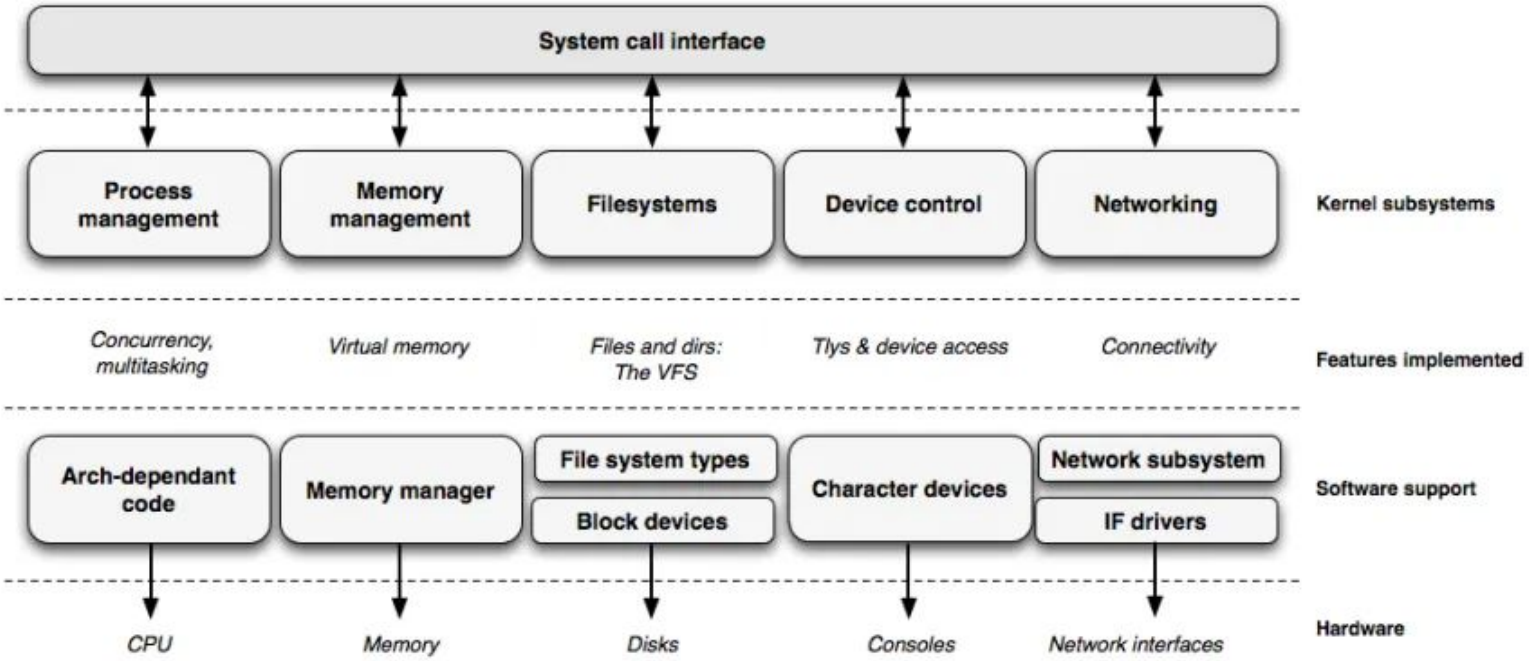
Unix toma la forma tradicional de un kernel, un programa especial que proporciona servicios a los programas en ejecución.



Linux: el Kernel

El kernel o núcleo es la barrera entre las aplicaciones de usuario y el hardware.

Linux: el Kernel



Requisitos Clave de un Sistema Operativo

Multiplexación

- El sistema operativo debe permitir que varios procesos se ejecuten simultáneamente, incluso si hay más procesos que CPUs disponibles.
- Esto implica compartir el tiempo y recursos de la computadora entre los procesos para asegurar que todos tengan la oportunidad de ejecutarse.

Aislamiento

- Debe asegurar que los procesos no afecten a otros en caso de errores. Si un proceso falla, no debe impactar a los procesos independientes.
- Sin embargo, este aislamiento no debe ser absoluto para permitir la interacción controlada entre procesos.

Requisitos Clave de un Sistema Operativo

Interacción

- Es fundamental que el sistema operativo permita la comunicación intencionada entre procesos.
- **Ejemplo:** Las tuberías (pipes) permiten que la salida de un proceso se convierta en la entrada de otro, facilitando la colaboración entre ellos.

Conclusión

- Para cumplir con estos requisitos, el sistema operativo debe manejar eficientemente la multiplexación, asegurar un adecuado aislamiento, y facilitar la interacción segura y controlada entre procesos.

Caso de Estudio: Unix

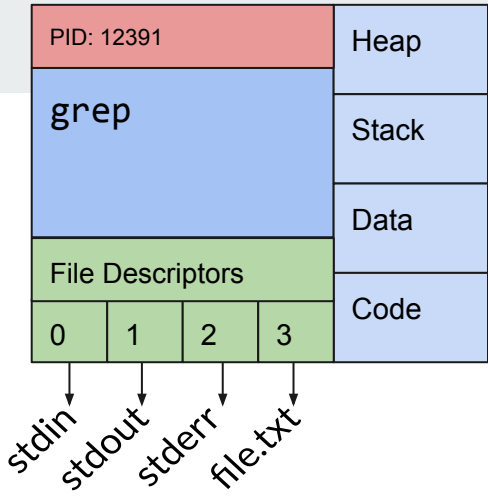
El diseño del sistema operativo Unix

Conceptos clave:

- Proceso
- File descriptors
- System calls: API de procesos

Procesos

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



Procesos

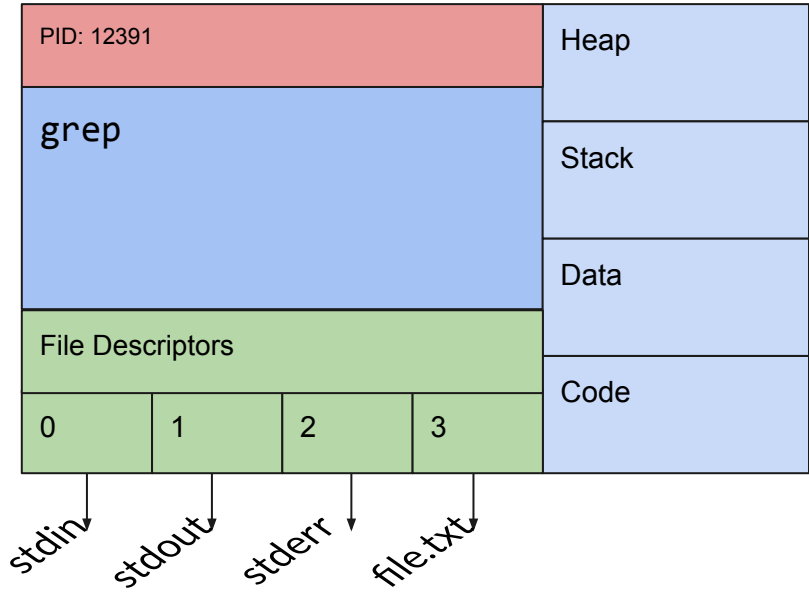
Partes básicas de un proceso:

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

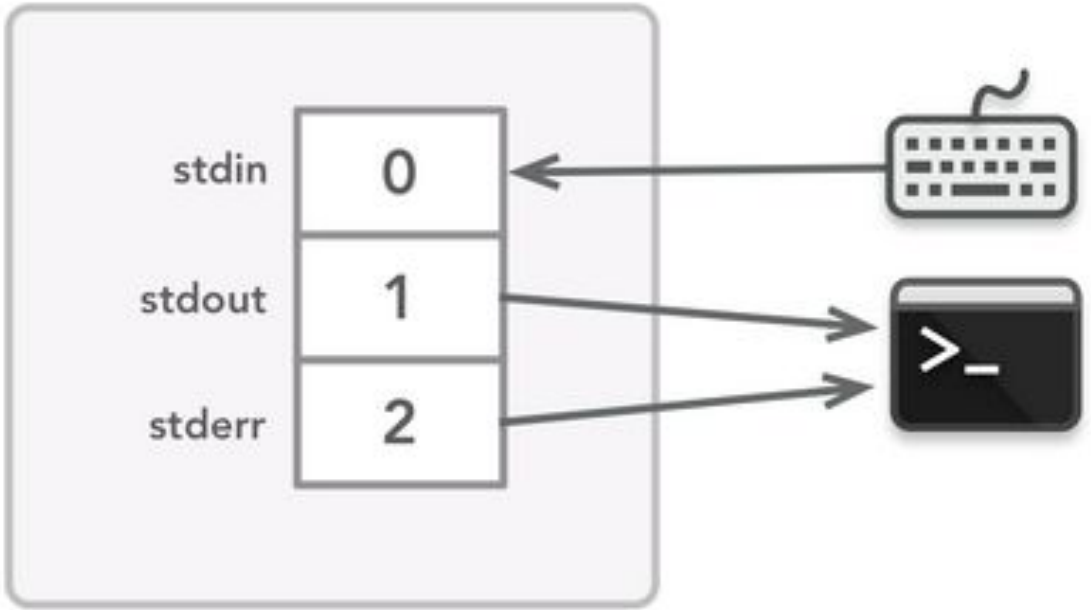
Los procesos tiene asociados los archivos abiertos que están usando. Estos archivos se identifican por un **número** denominado **file descriptor,** un número entero positivo.

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

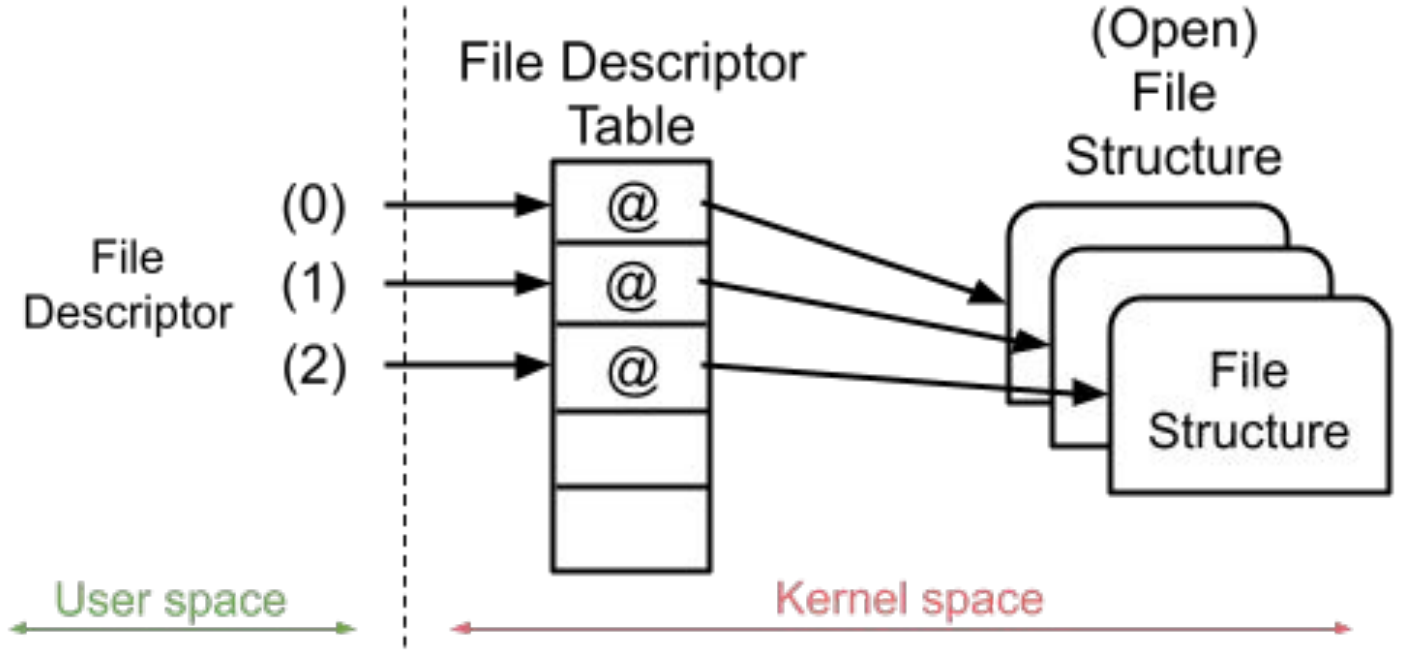
Existen 3 file descriptors que son creados cuando se crea un proceso:

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 de Procesos Resumida

fork(): Crea un nuevo proceso y devuelve su ID (pid). El nuevo proceso es una copia del proceso actual.
wait(): Suspende la ejecución del proceso actual hasta que uno de sus procesos hijos termine. Devuelve el ID del proceso hijo terminado.
getpid(): Devuelve el ID del proceso actual, permitiendo su identificación dentro del sistema.
exec(filename, argv): Carga y ejecuta un nuevo programa, reemplazando el proceso actual con el contenido del archivo especificado. Los argumentos para el nuevo programa se pasan a través de argv.
exit(): Finaliza el proceso actual, devolviendo un código de salida al sistema operativo.
kill(pid): Envía una señal para terminar el proceso especificado por su ID (pid), permitiendo la terminación controlada de procesos.
pipe(): Crea un canal de comunicación entre procesos, permitiendo que uno lea datos por un extremo y el otro escriba datos por el otro extremo.
dup(): Duplica un descriptor de archivo, creando una copia exacta de él. Esto es útil para redirigir la entrada o salida de un proceso.

System Call fork()

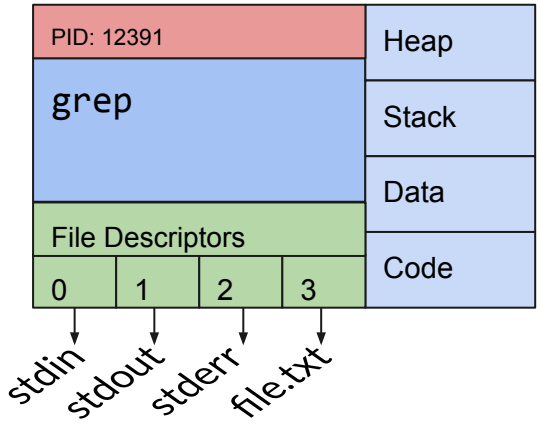
#include <unistd.h>
pid_t fork(void);

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.

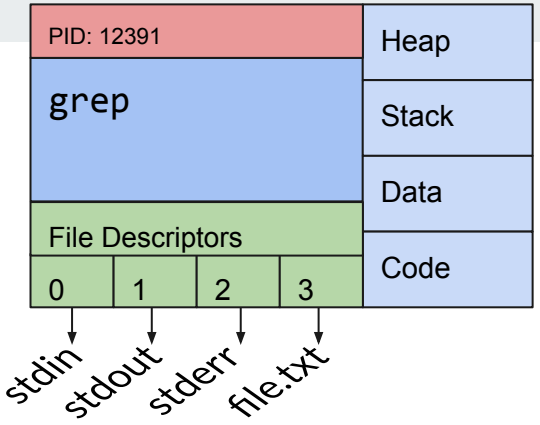
System Call fork()



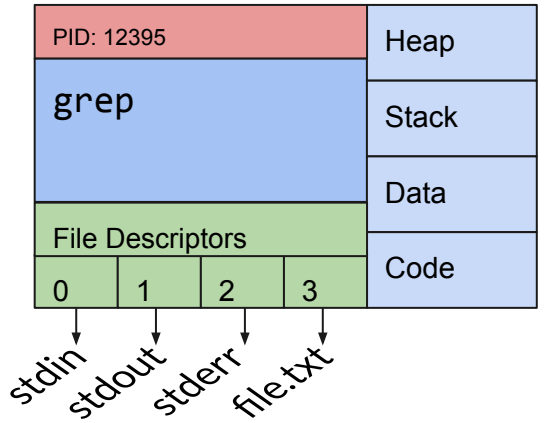
pid:12391

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 direccion 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 unicamente tuvieramos 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()

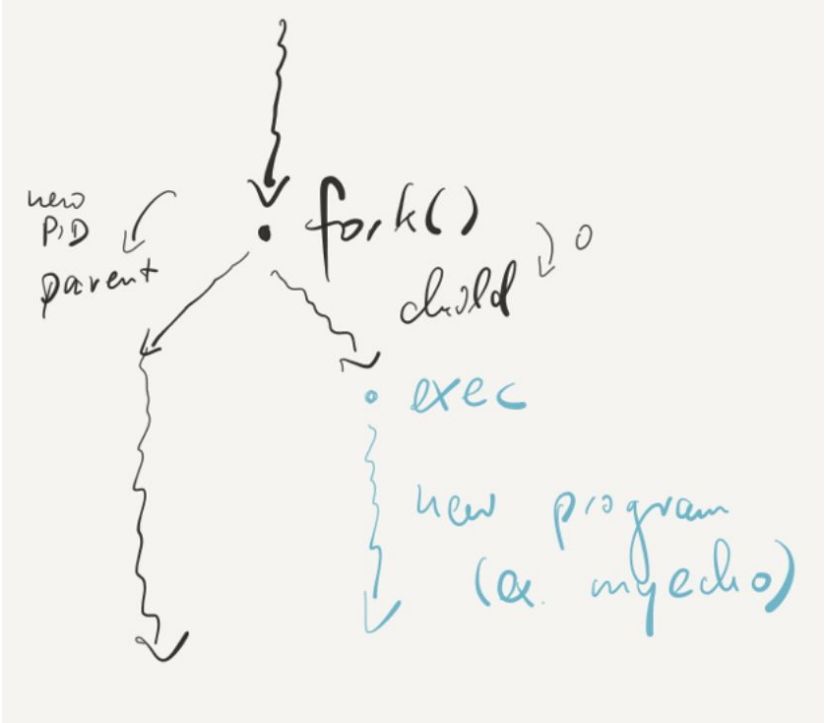
Ojo que no se crea ningún proceso solo se reemplaza
Un programa en ejecución por otro.

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()

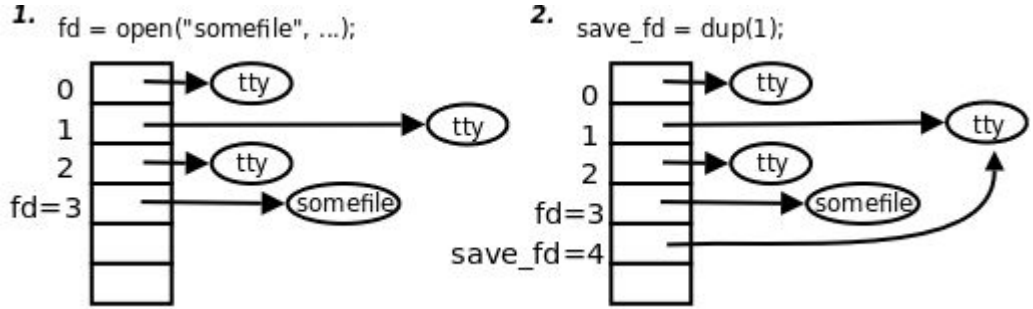
```
#include <unistd.h>
int dup(int fildes);
int dup2(int fildes, int fildes2);
```

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);

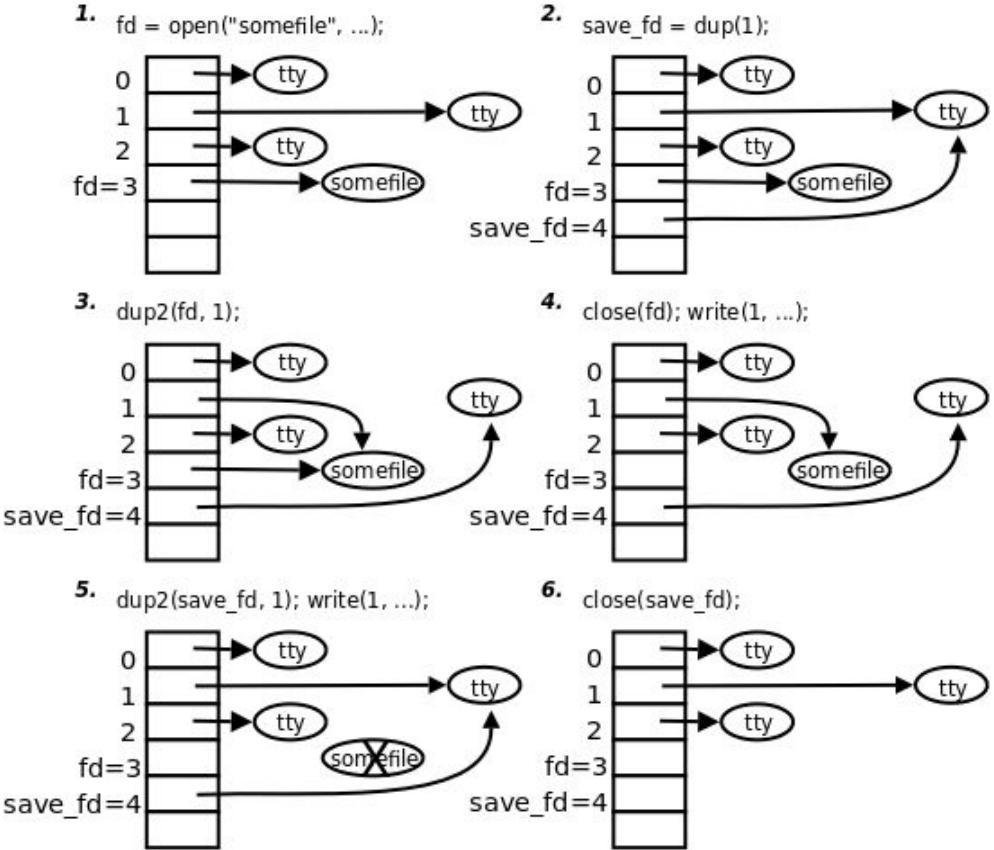
dup() retorna un nuevo file descriptor que es copia del enviado como parámetro, obteniendo el primer file descriptor libre que se encuentre en la **File Descriptor Table**.



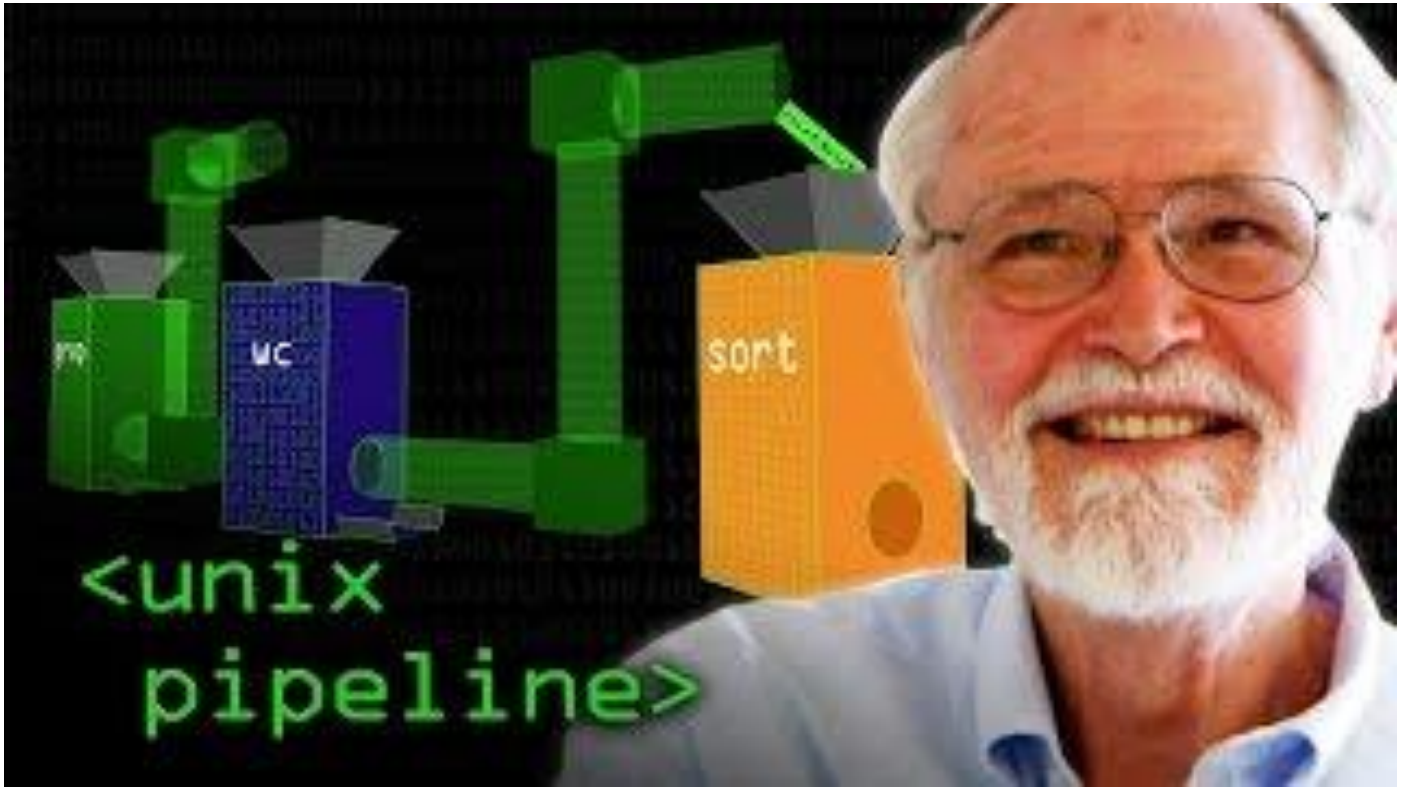
System Call dup()

```c
#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);
}
```

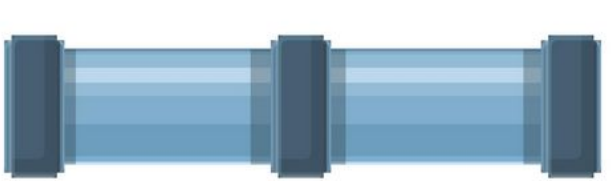
System Call dup()



Pipelines



System Call pipe()

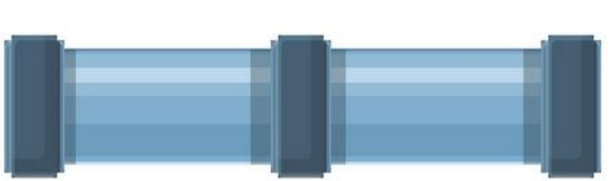


```
#include <unistd.h>
int pipe(int pipefd[2]);
```

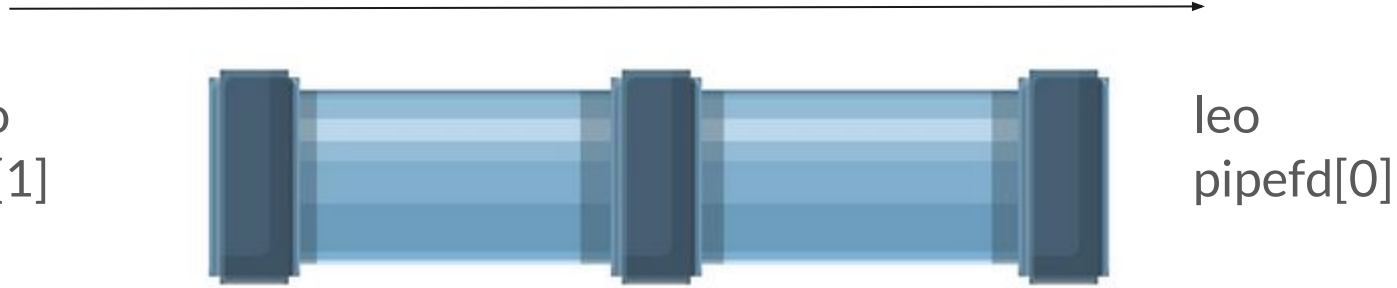
Un pipe es un **pequeño buffer en el kernel** expuesto a los procesos como un par de files descriptors, uno para escritura y otro para lectura. Al escribir datos en el extremo el pipe hace que estos estén disponibles en el otro extremo para ser leídos.

ES UN CANAL UNIDIRECCIONAL!!!

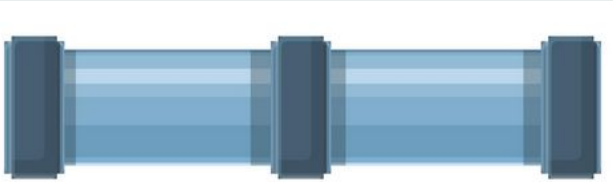
System Call pipe()



Int pipefd[2]

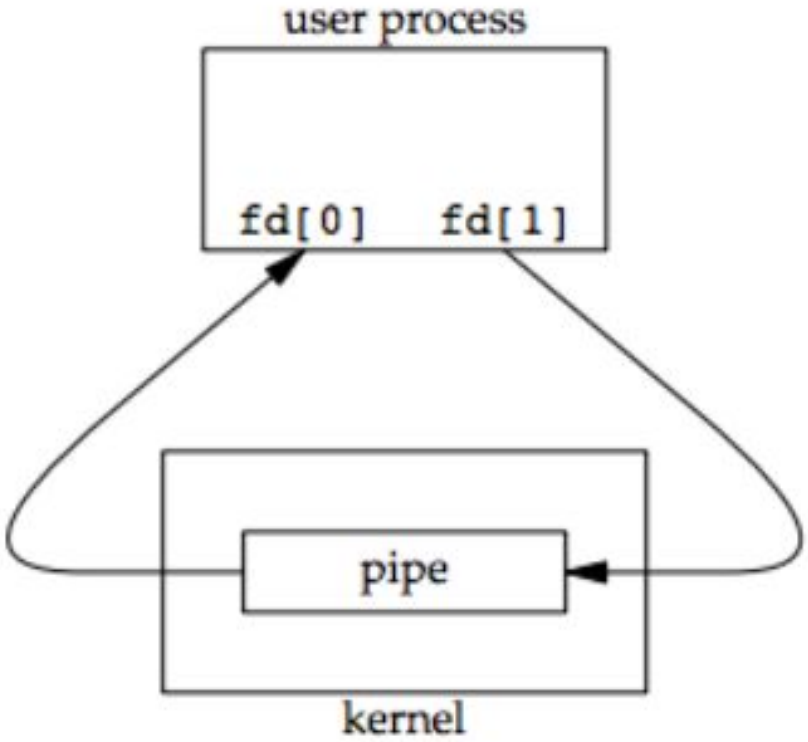


System Call pipe()

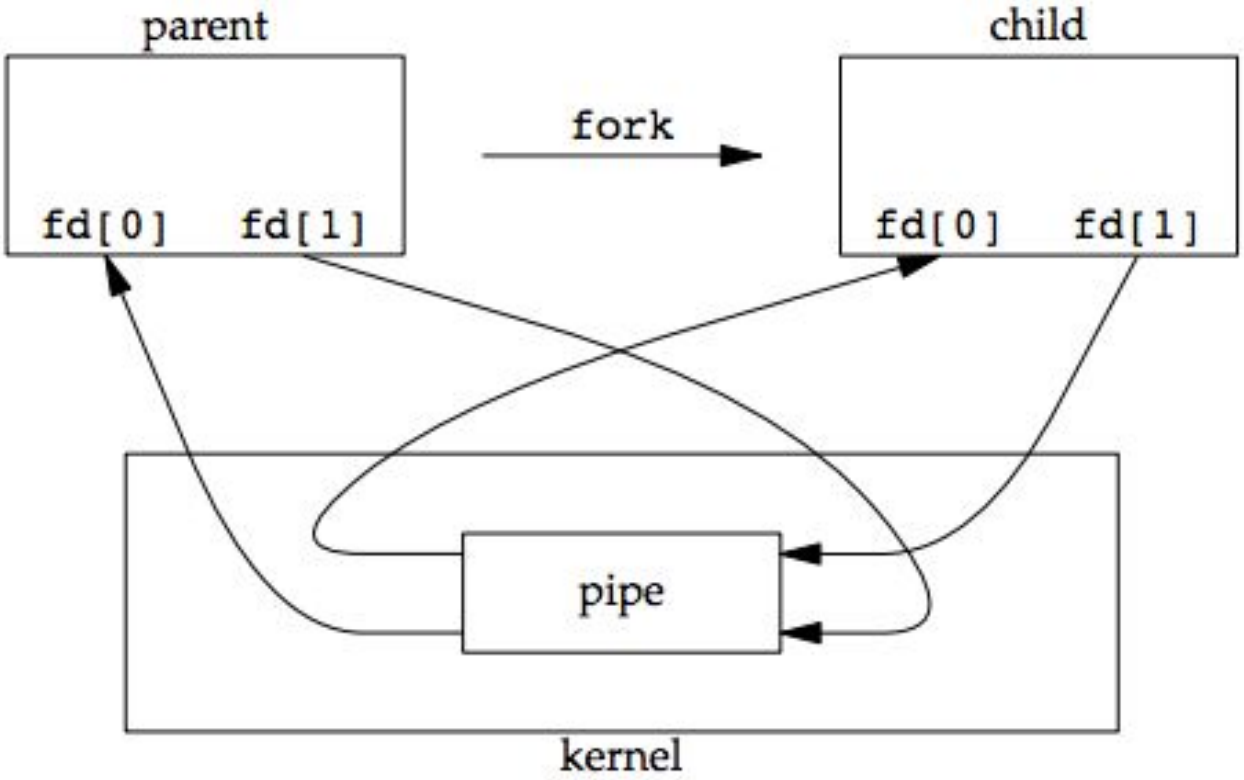


Puedo hacer que dos procesos compartan información.

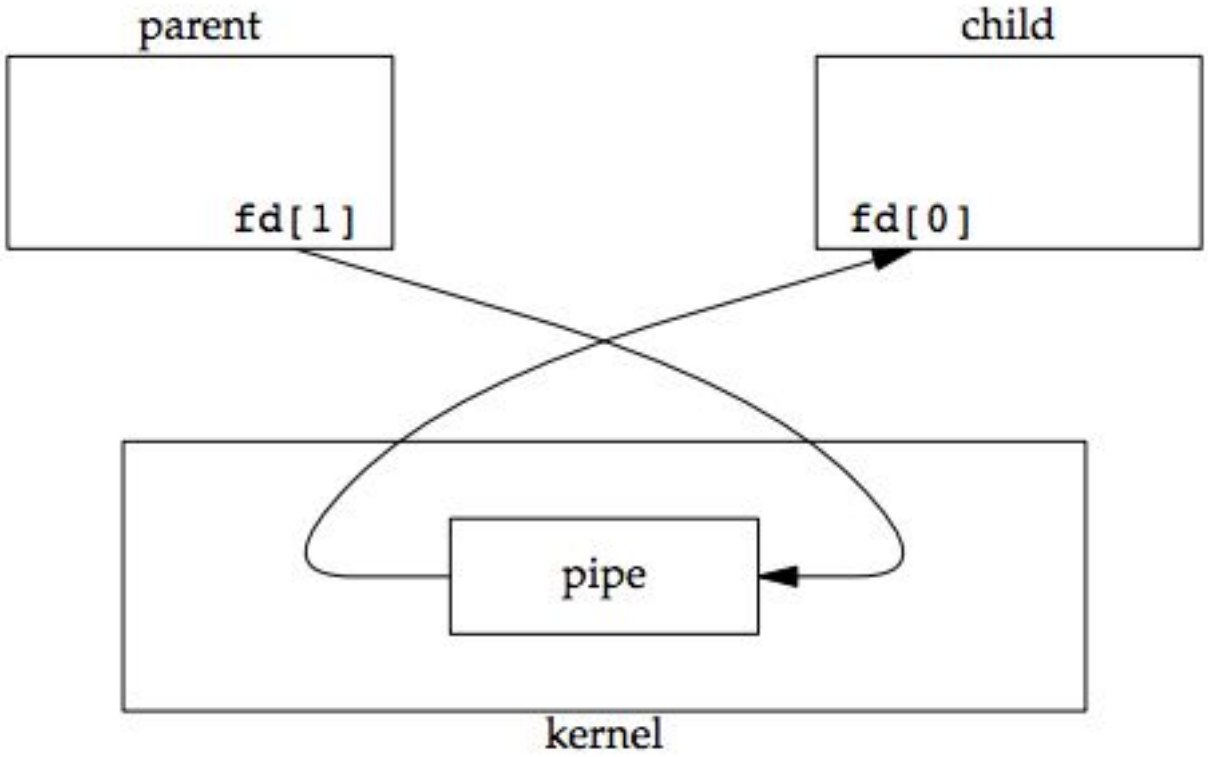
System Call pipe()



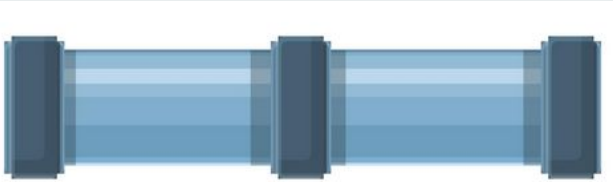
System Call pipe()



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]);
}
```

User/Kernel

Conceptos clave:

- Aislamiento de los programas
- Protección por hardware
- System calls

User Space

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.

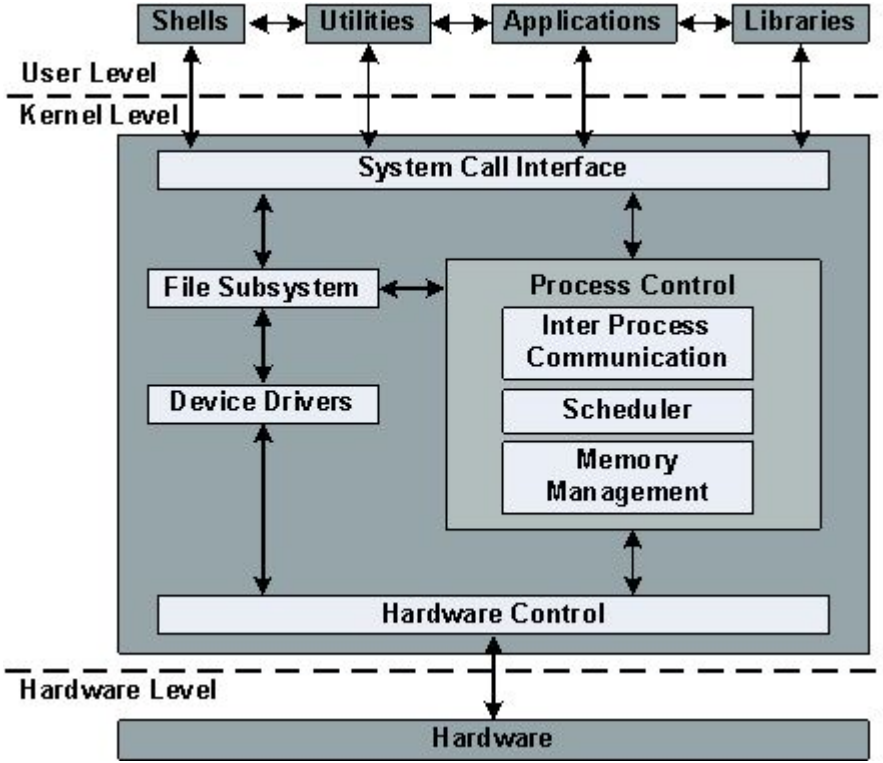
¿Y qué pasa en User space?

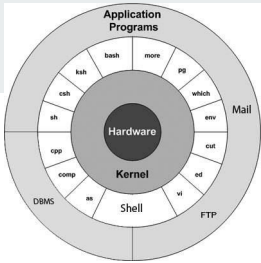
Los programas se están ejecutando!

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.

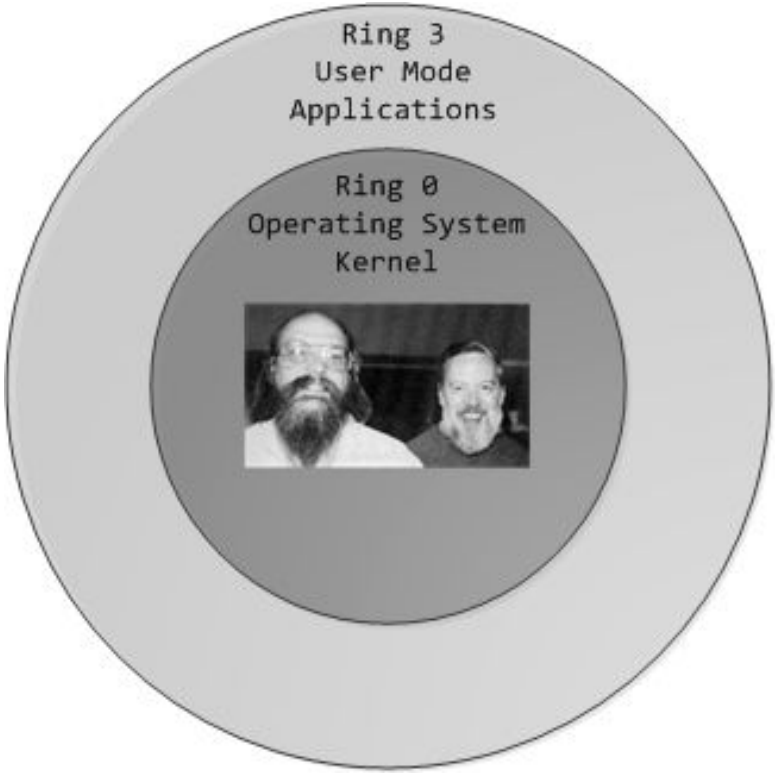
El Kernel

Kernel-space





User-space vs Kernel space



Las CPUs proporcionan soporte de hardware para un aislamiento fuerte. Por ejemplo:

RISC-V tiene tres modos en los que la CPU puede ejecutar instrucciones: **modo máquina, modo supervisor y modo usuario**.

X86 tiene cuatro modos en los que la CPU puede ejecutar instrucciones, solo usa 2: **modo supervisor y modo usuario**.

En modo **supervisor**, la CPU puede ejecutar instrucciones privilegiadas: por ejemplo, habilitar y deshabilitar interrupciones, leer y escribir en el registro que contiene la dirección de una tabla de páginas, etc.

User-space vs Kernel space

Una aplicación solo puede ejecutar instrucciones en modo usuario (por ejemplo, sumar números, etc.) y se dice que se ejecuta en el espacio de usuario, mientras que el software en modo supervisor también puede ejecutar instrucciones privilegiadas y se dice que se ejecuta en el espacio del kernel.

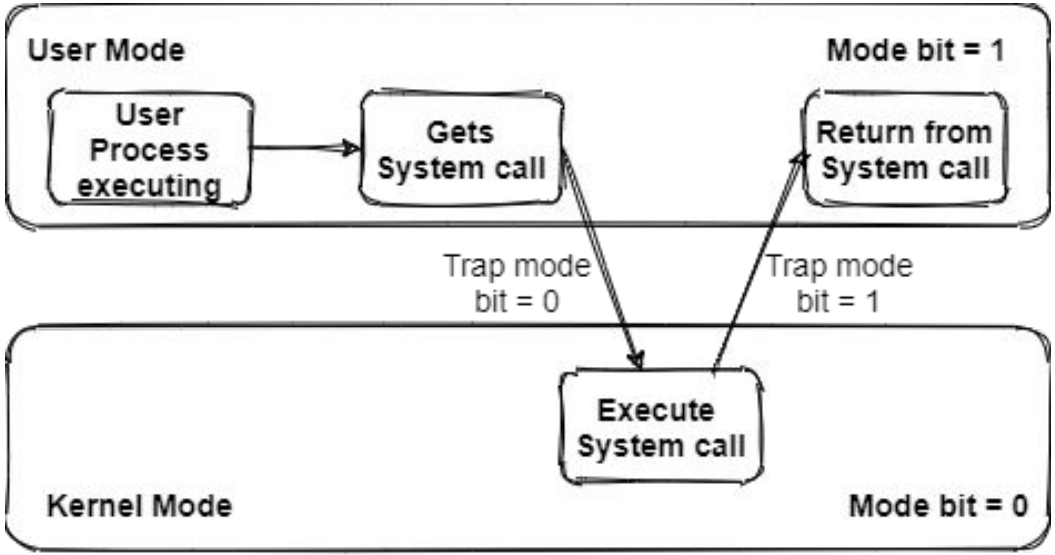


Image by https://cplusplus.in/

User-space vs Kernel space

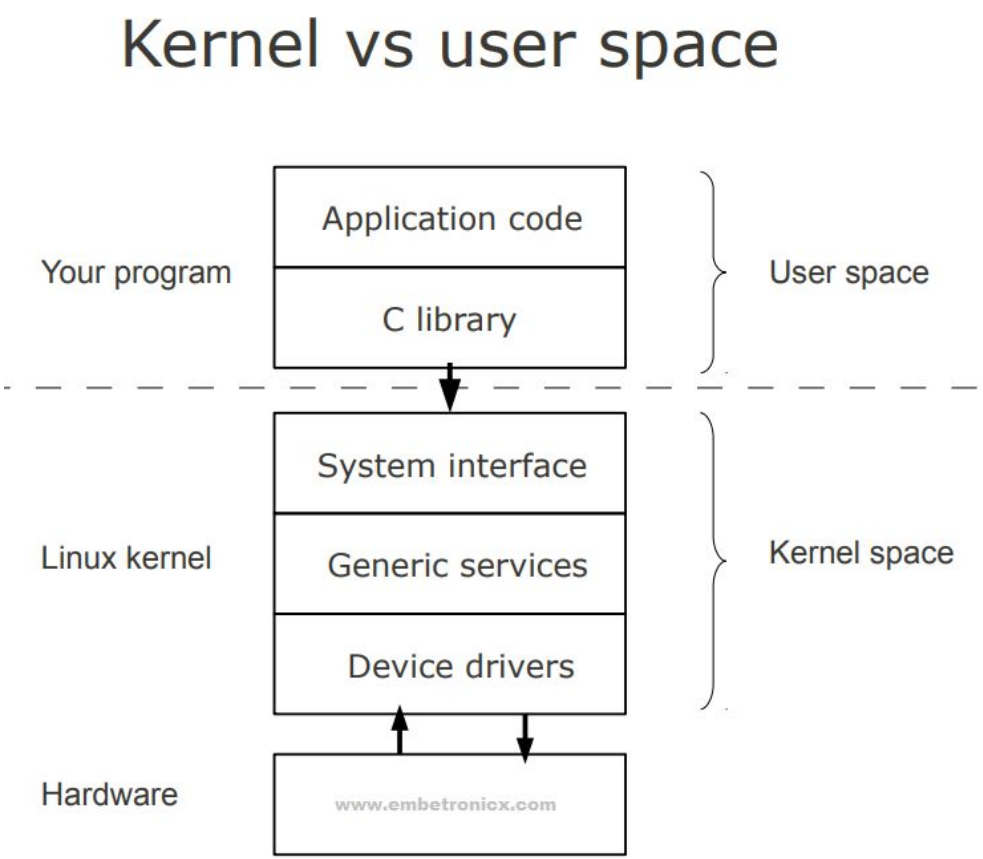
Una aplicación que quiere invocar una función del Kernel (por ejemplo, la syscall read en xv6) debe realizar una transición al Kernel; una aplicación no puede invocar directamente una función del Kernel.

Las CPUs proporcionan una instrucción especial que cambia la CPU del modo usuario al modo supervisor y entra en el Kernel en un punto de entrada especificado por el Kernel. (RISC-V proporciona la instrucción ecall para este propósito, x86 int 0x80.)

Una vez que la CPU cambio al modo supervisor, el Kernel puede entonces validar los argumentos de la llamada al sistema (por ejemplo, verificar si la dirección pasada a la llamada al sistema es parte de la memoria de la aplicación), decidir si la aplicación tiene permisos para realizar la operación solicitada (por ejemplo, verificar si la aplicación tiene permisos para escribir en el archivo especificado) y luego denegarla o ejecutarla.

Es importante que el Kernel controle el punto de entrada para las transiciones al modo supervisor; Si la aplicación pudiera decidir el punto de entrada al núcleo, una aplicación maliciosa podría, por ejemplo, ingresar al núcleo en un punto donde se omite la validación de argumentos.

User-space vs Kernel space



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**.

Una Syscall es un servicio que provee el kernel

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.