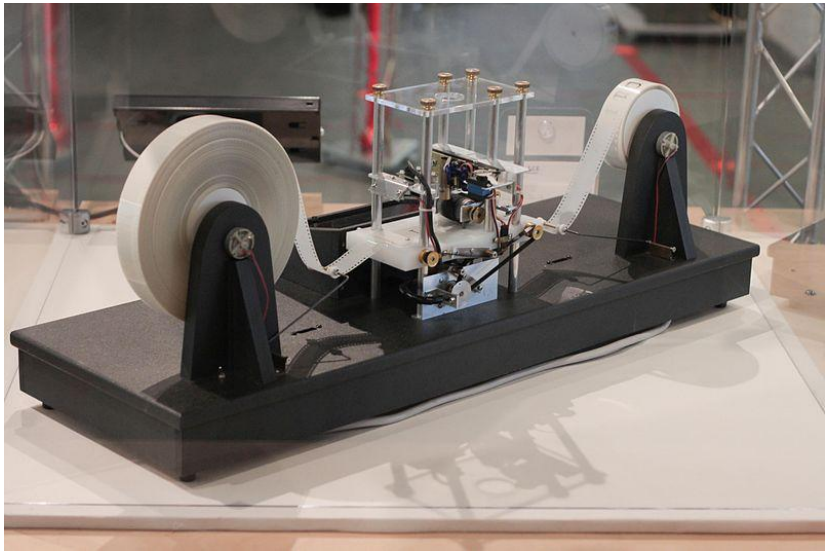
Sistemas Operativos
Fisop 2025

Introducción

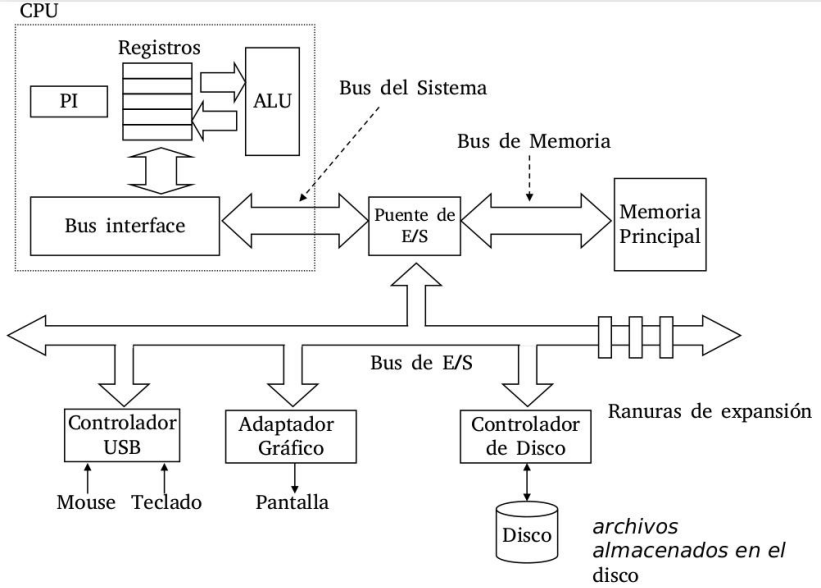
¿Qué es un Sistema Operativo?

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

Alan Turing



John Von Neuman



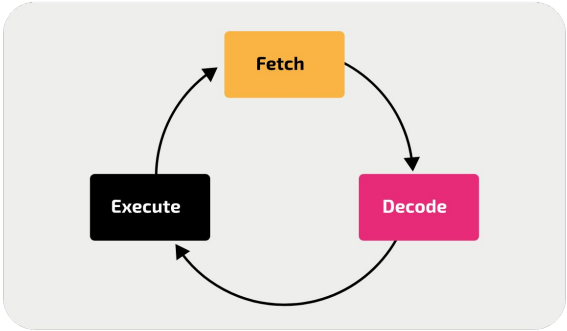
¿Qué pasa cuando un programa se ejecuta?

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

La decodifica (**decode**)

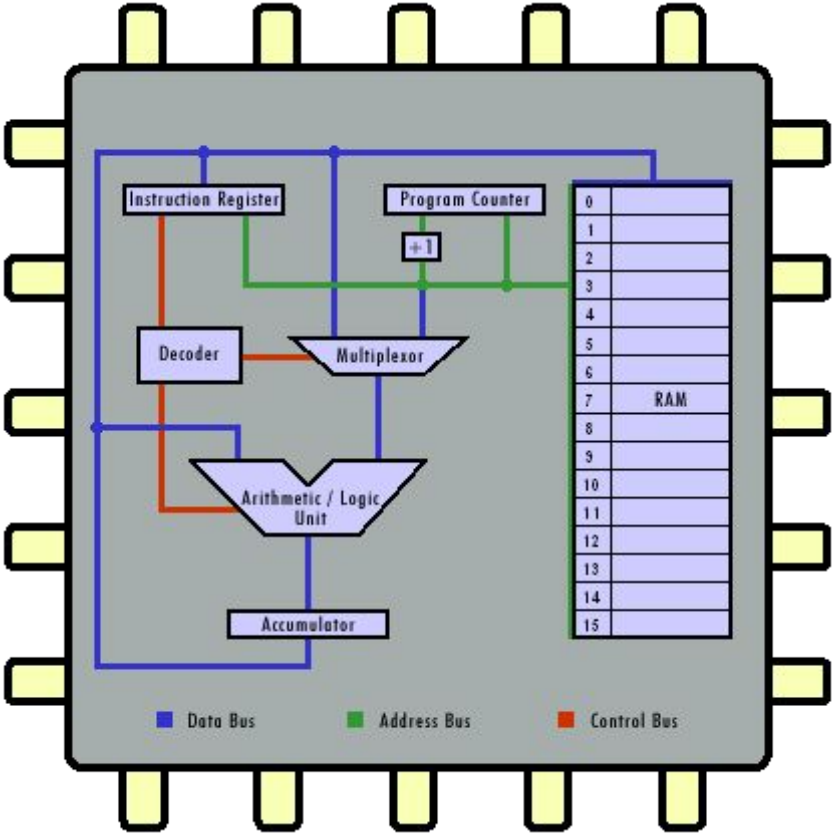
Finalmente la ejecuta (**execute**)

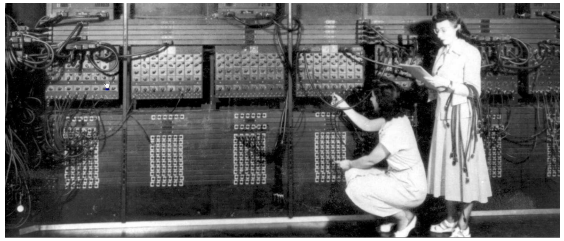
fetch-decode-execute



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

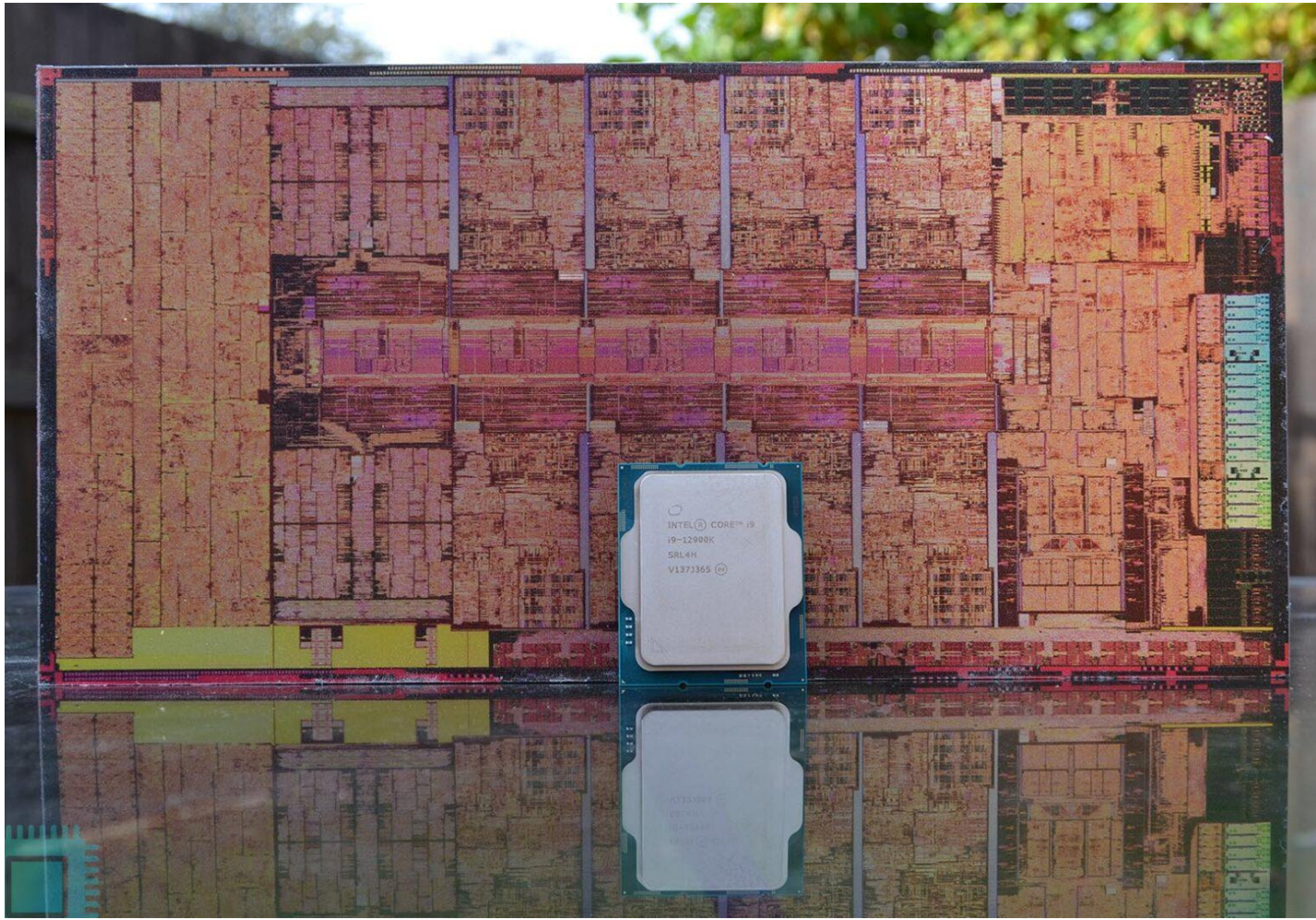
Clementina (Ferranti Mercury) - 1960 - Pabellón I Ciudad Universitaria

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."



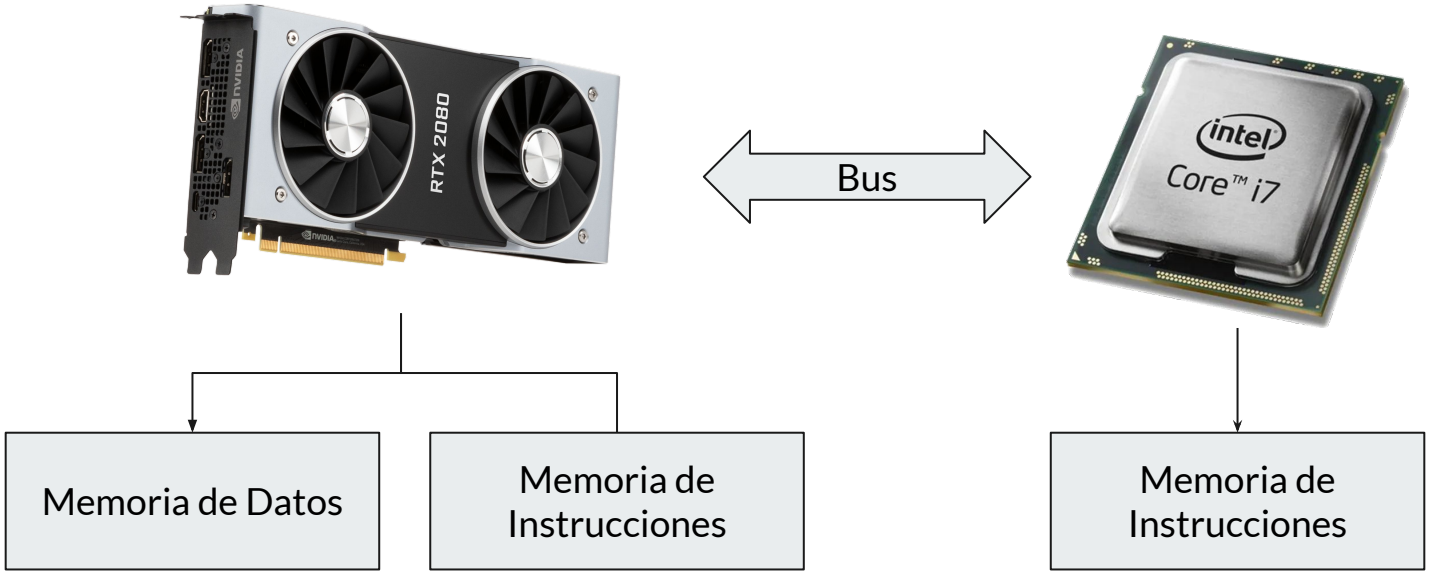
Tendencias actuales

GPUs

- Se usa para computación en paralelo.
- Calculo cientifífico
- Entrenamiento de Redes Neuronales

Tendencias actuales

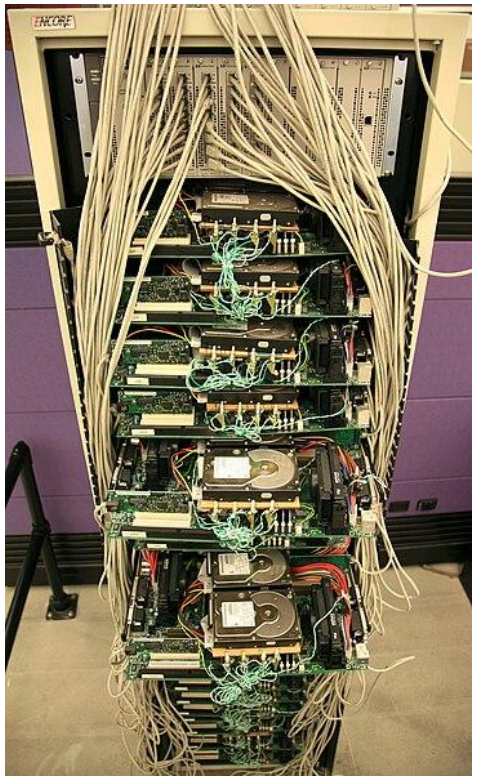
GPUs



Ya no es una arquitectura de Von Neuman!

Tendencias actuales

Cloud Computing

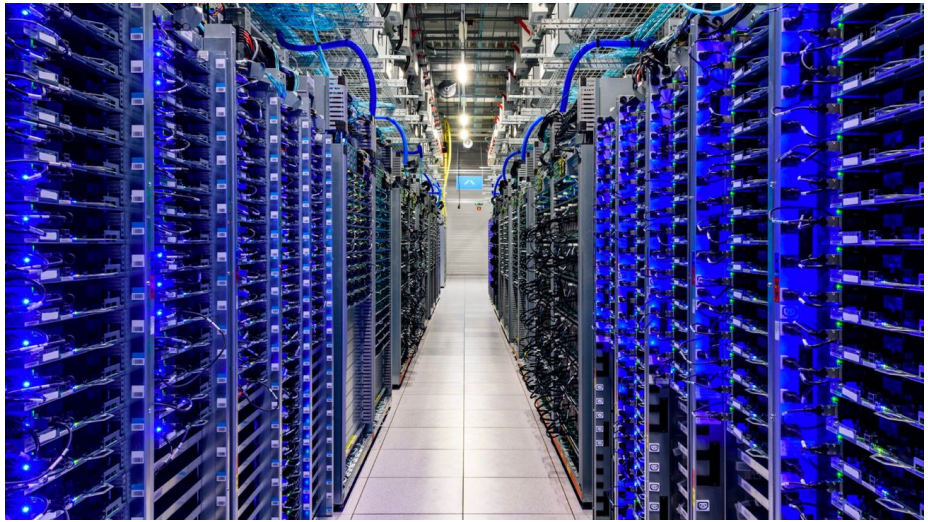


Primer servidor de produccion de Google

- Surge de la idea de armar una “super computadora” con commodity hardware
- Usa tecnologías de virtualización y contenedorización

Tendencias actuales

Cloud Computing

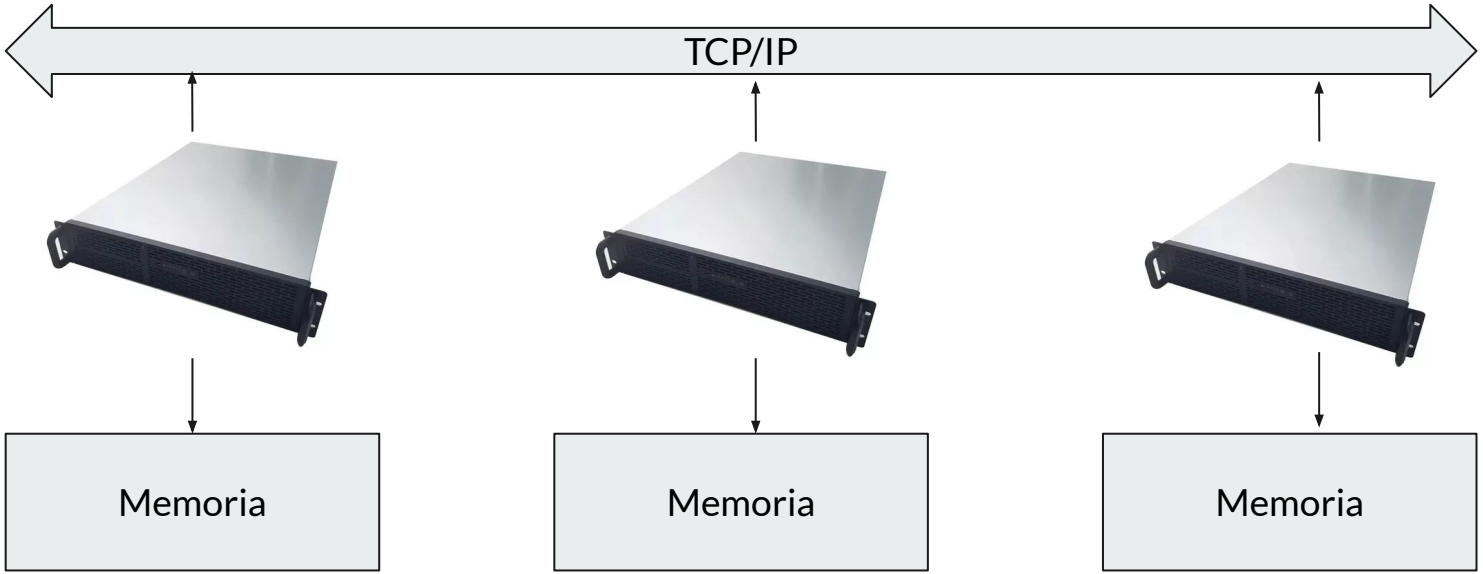


- Surge de la idea de armar una “super computadora” con commodity hardware
- Usa tecnologías de virtualización y contenedorización

Asi se ve ahora

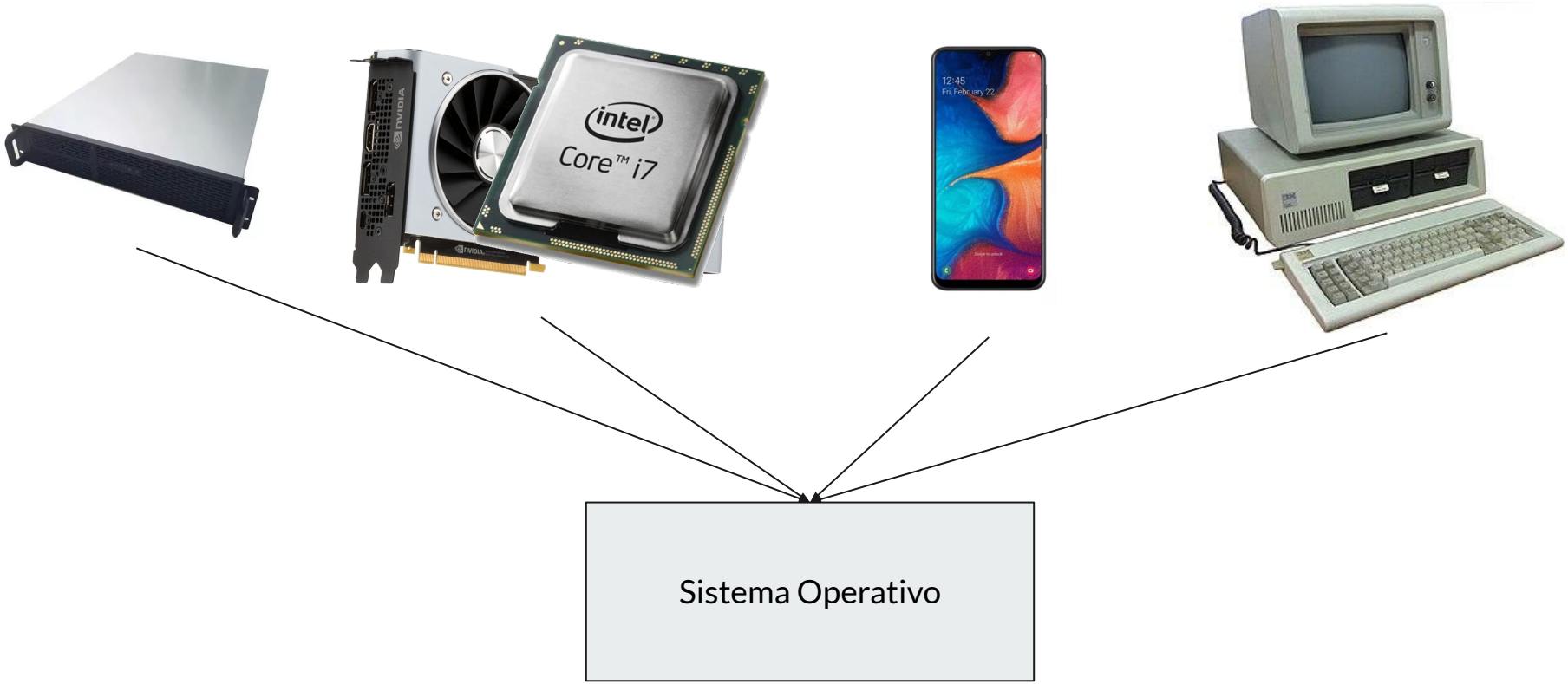
Tendencias actuales

Cloud Computing

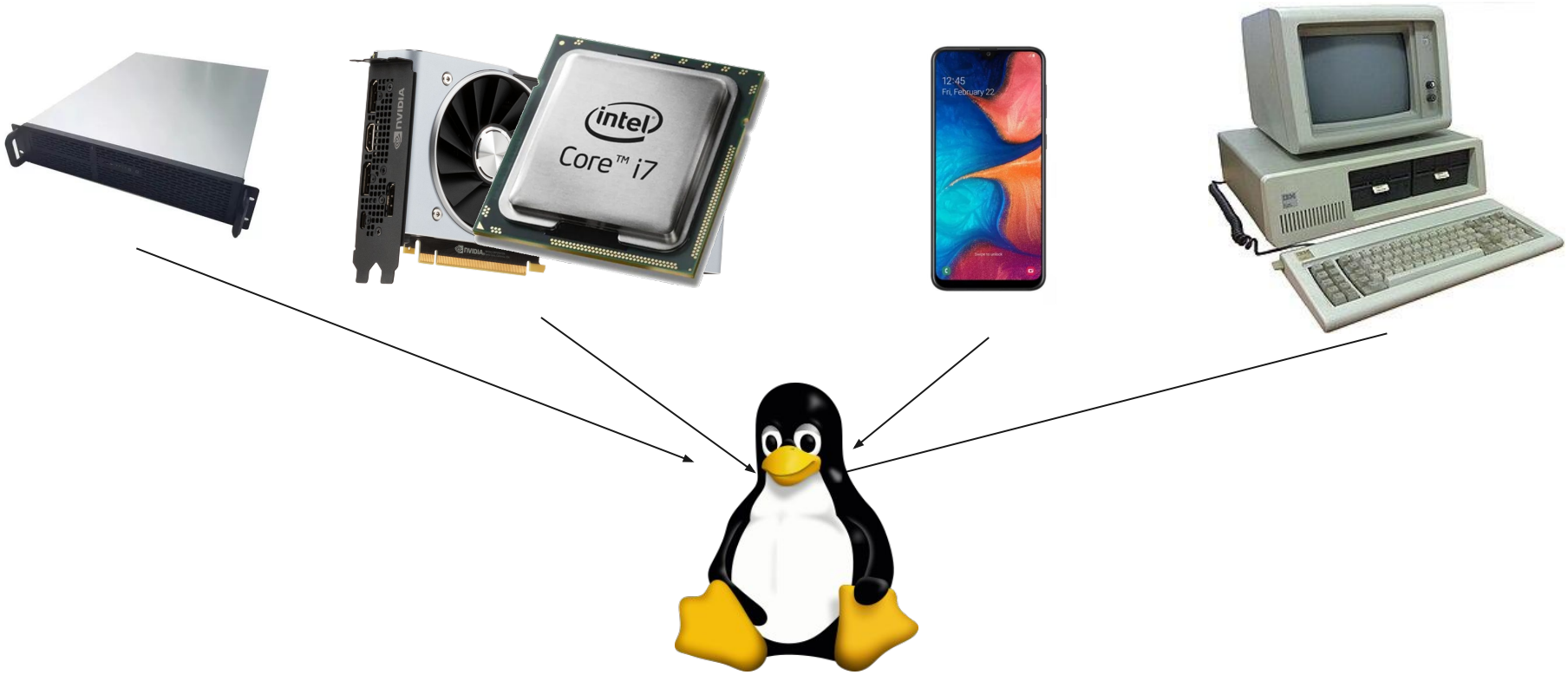


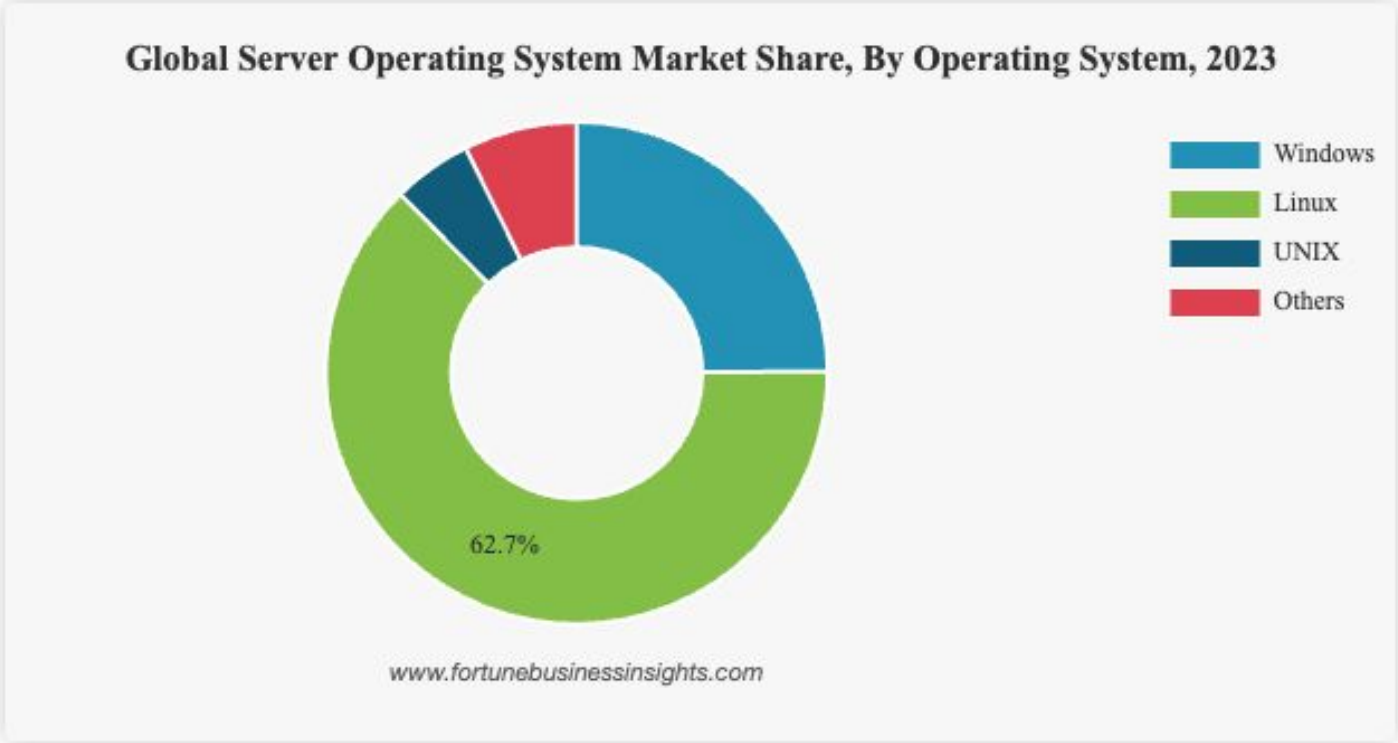
Ya no es una arquitectura de Von Neuman!
(esto es un Sistema Distribuido)

Qué tienen todos en común?



Qué tienen todos en común?





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

Motivación

Rol del sistema operativo moderno

Conceptos clave:

- Definiciones fundamentales
- Funciones del SO
- Virtualización

¿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:

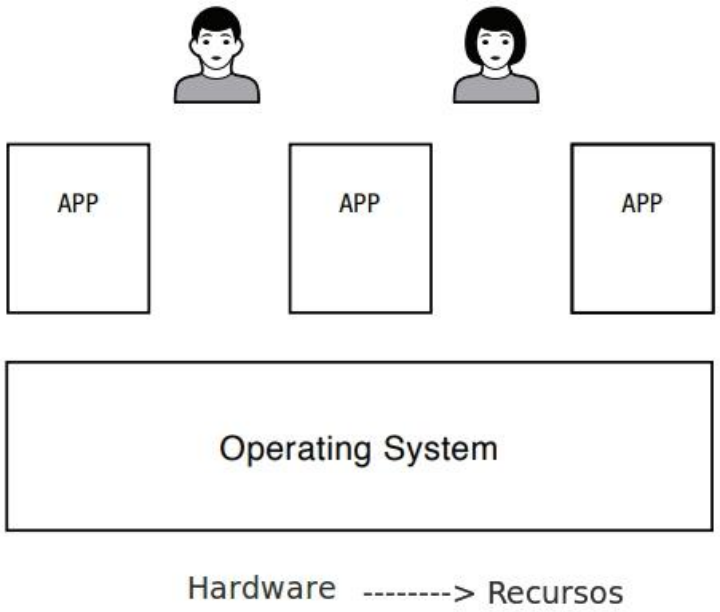
- Por un lado, nos gustaría que la interfaz fuera **simple y reducida** porque eso facilita hacer que la implementación sea correcta.
- Por otro lado, podríamos estar tentados a ofrecer muchas características sofisticadas a las aplicaciones. El truco para resolver esta tensión es diseñar interfaces que 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.

Estas 3 metáforas nos permiten entender qué funciones realiza un SO

- Referee (Árbitro)
- Ilusionista
- Glue (conector)

Funciones de un S.O.

Referee (Árbitro)

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.

Funciones de un S.O.

Referee (Árbitro)

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.

Funciones de un S.O.

Glue (Conector)

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

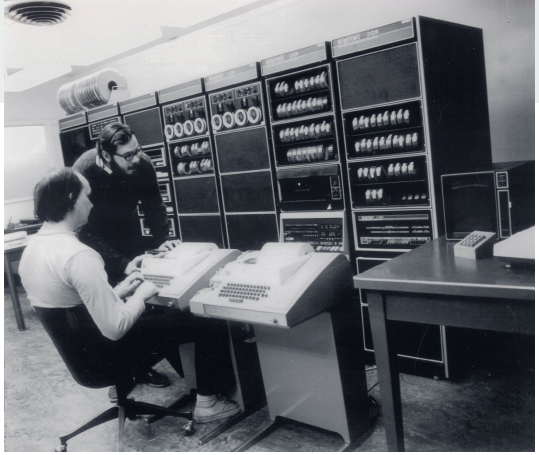
Unix

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.

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:

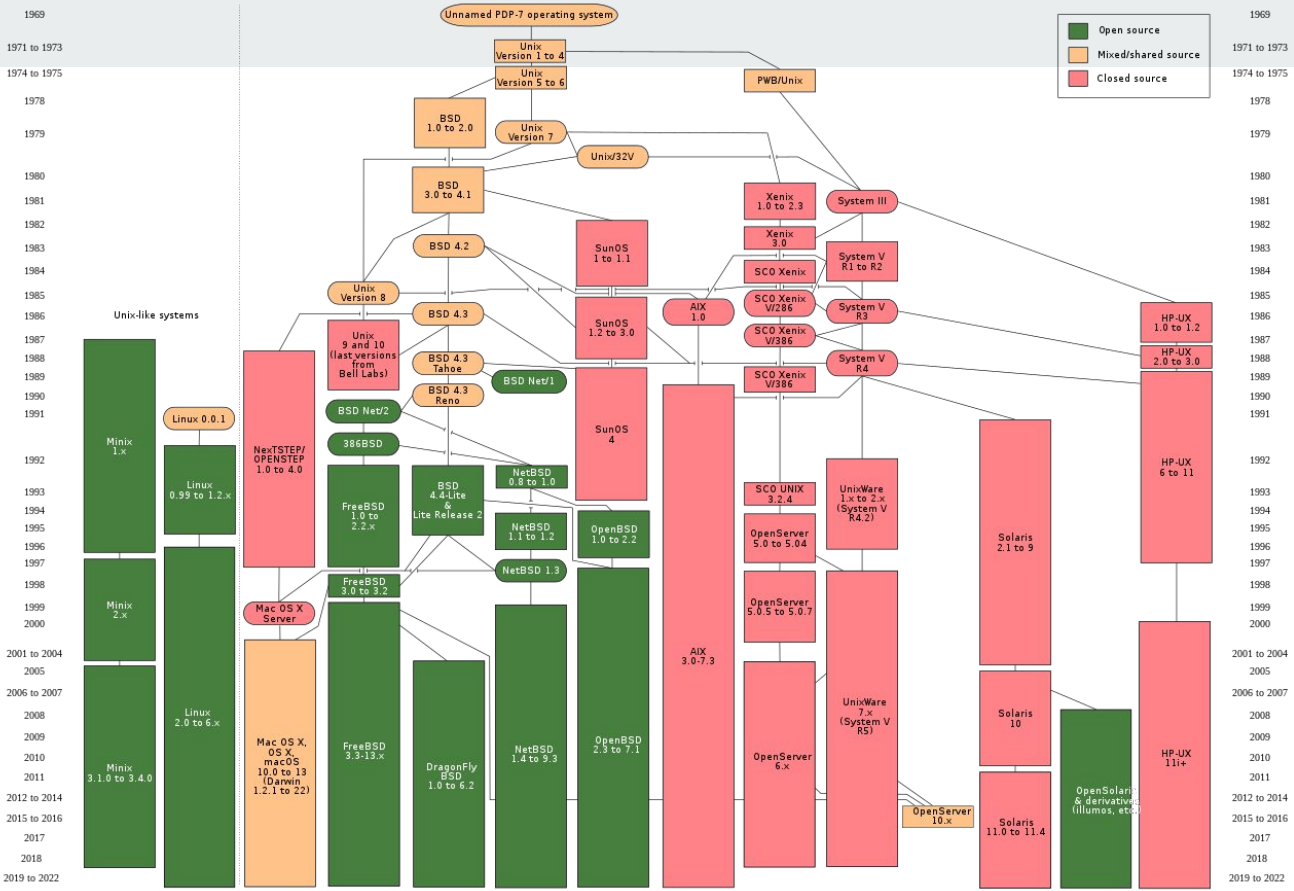
- Diseña la **salida de cada programa** para que pueda ser utilizada como **entrada de otros programas**. Evita formatos complejos o interactivos.

Pruebas Tempranas y Refactorización:

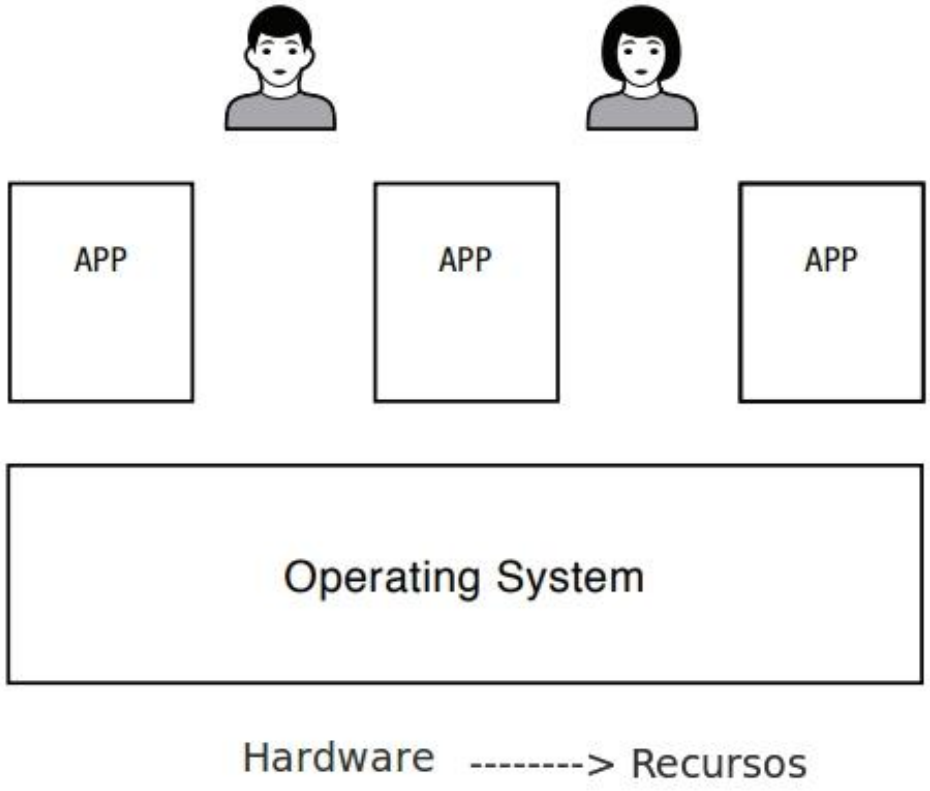
- Diseña software, incluidos sistemas operativos, para ser probado temprano, idealmente en semanas. No dudes en descartar y rehacer partes ineficaces.

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.

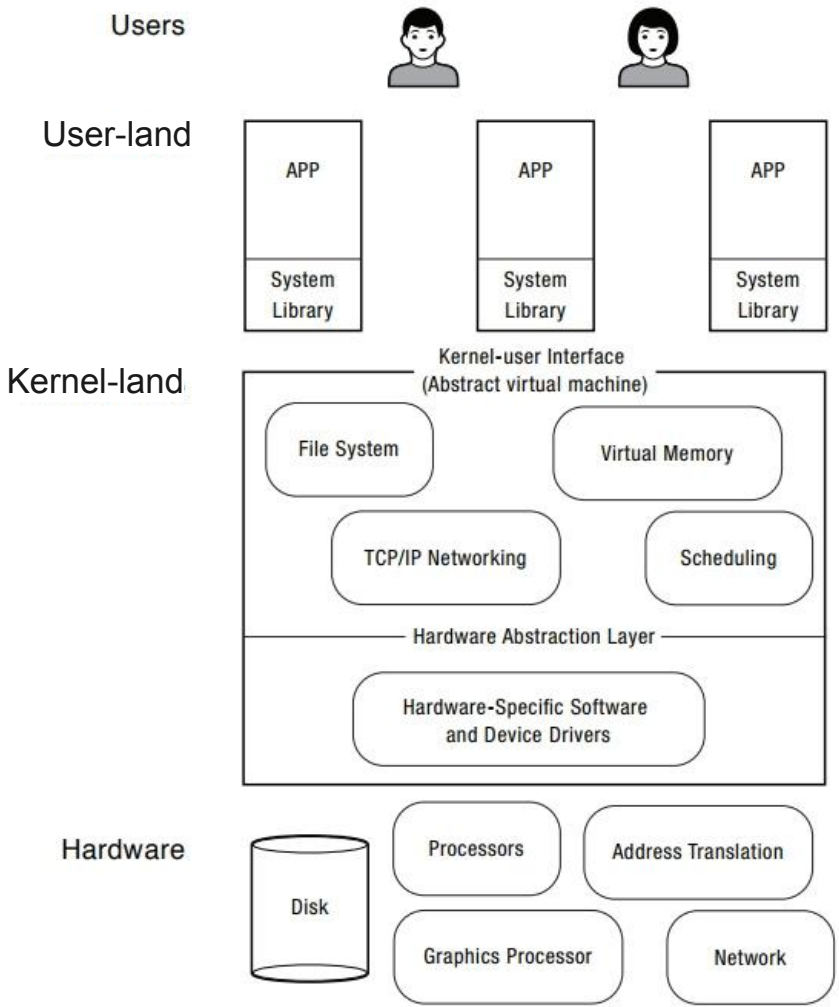


Unix Evolution



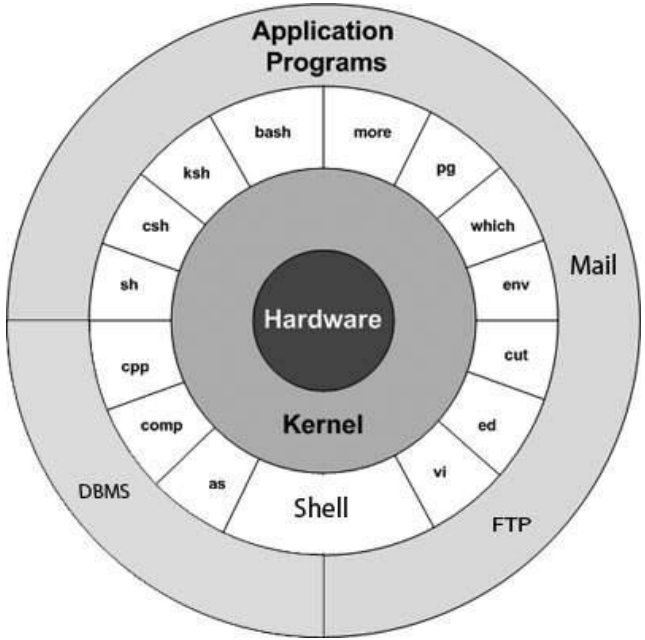
Sistema Operativo

Sistema Operativo Unix-like

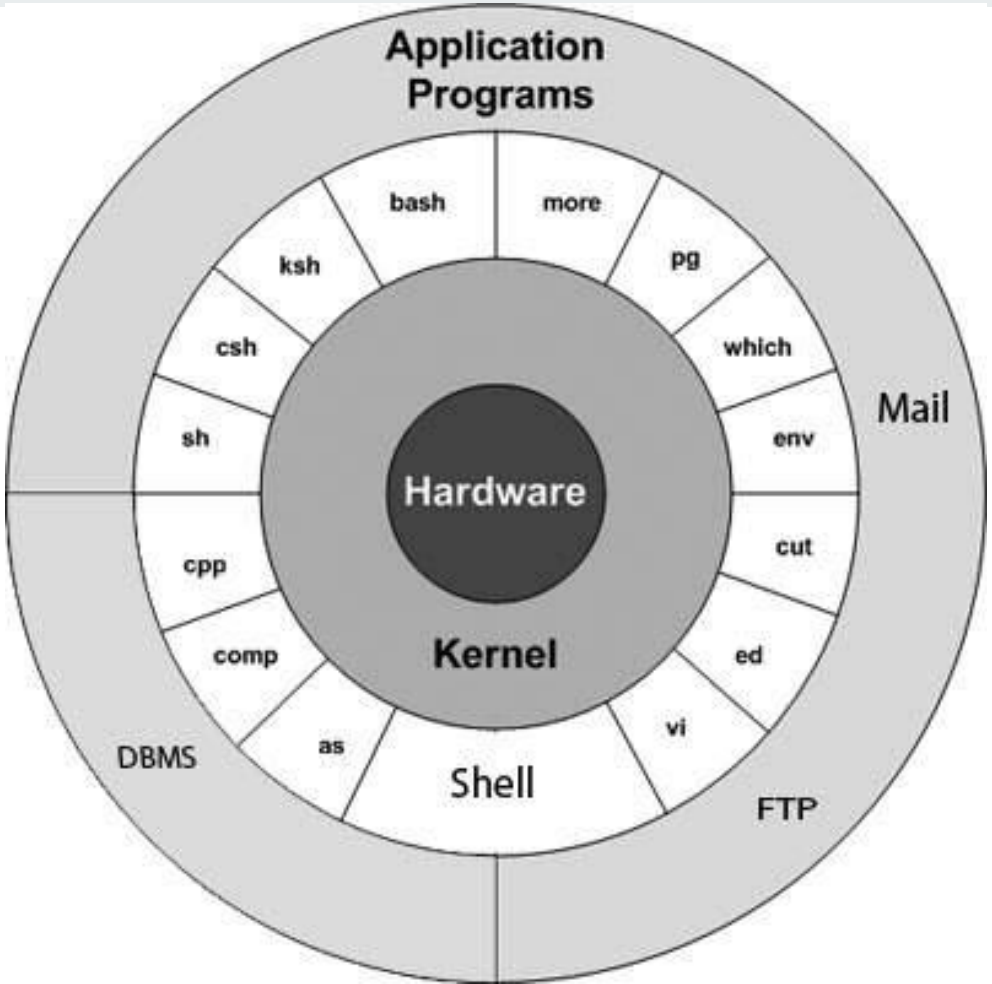


Linux: el Kernel

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



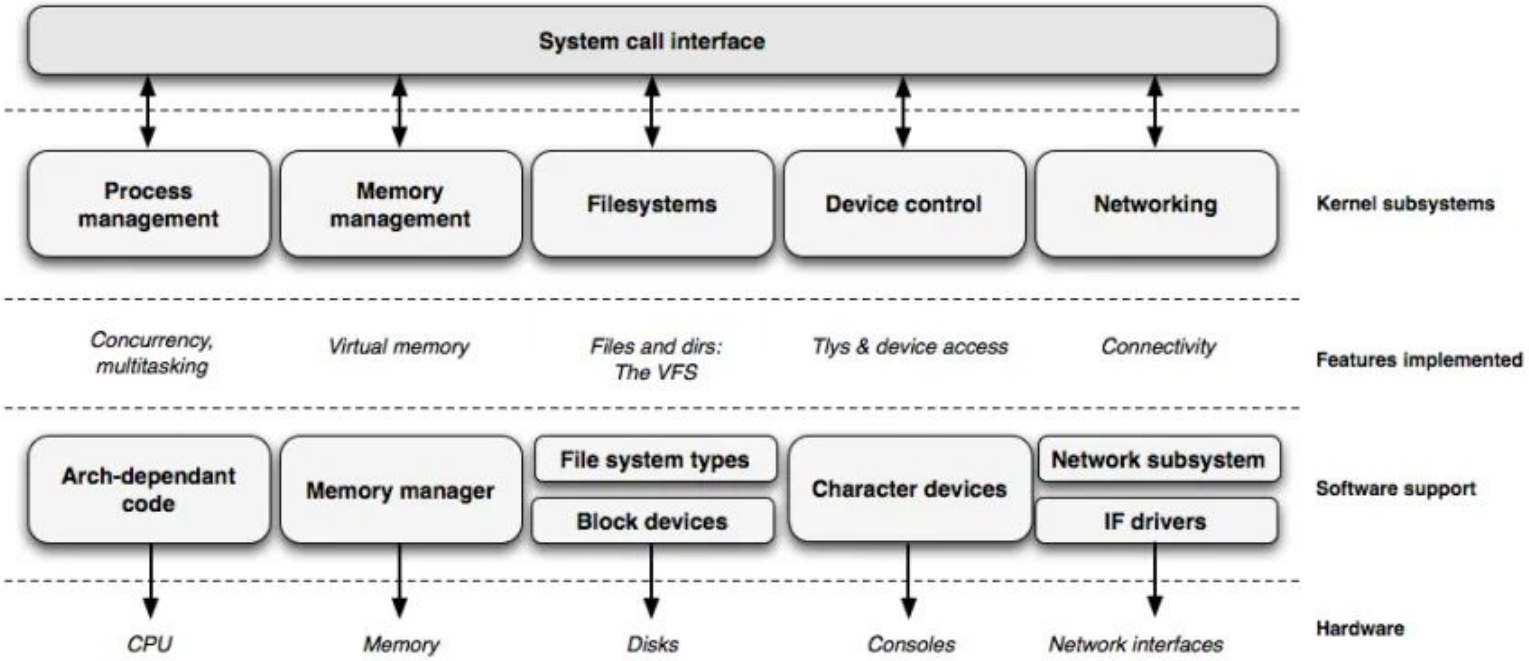
Linux: el Kernel



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.

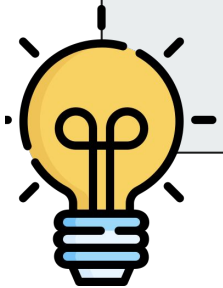
Caso de Estudio: Unix

El diseño del sistema operativo Unix

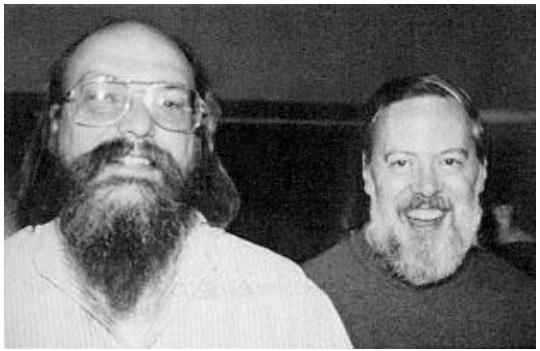
Conceptos clave:

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

open(), close(), read(), write()



En Unix (casi) todo es un archivo



Ken Thompson y Dennis ritchie
Creadores de Unix

Si casi todo es un archivo, podemos hacer casi todo con una API de archivos:

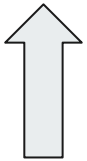
- open()
- close()
- read()
- write()

open(), close(), read(), write()

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

int main() {
    int fd = open("salida.txt", O_WRONLY | O_CREAT |
O_TRUNC, 0644);
    if (fd == -1) return 1;

    write(fd, "Hola, mundo\n", 12);
    close(fd);

    return 0;
}
```

open(), close(), read(), write()

```
int fd = open("salida.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);
```



Esto es un **File Descriptor**

Cuando un programa abre un archivo (o cualquier recurso similar), el sistema operativo le asigna un número entero llamado file descriptor, que se usa para leer, escribir o cerrar el archivo.

open(), close(), read(), write()

Ken Thompson (más viejo)

Cosas que puede representar un file descriptor

1. **Archivos normales**
2. **Dispositivos de entrada/salida (I/O)**
   - Eg. `/dev/sda`, `/dev/nvme0n1`)
3. **Sockets de red**
   - Conexiones TCP y UDP (`socket()` devuelve un file descriptor)
4. **Pipes y FIFOs (Named Pipes)**
   - Comunicación entre procesos en la misma máquina (`pipe()`, `mkfifo()`).
5. **Pseudoterminales (PTY)**
   - Usadas en emuladores de terminal como `tmux` o `screen` (`/dev/pts/X`).
6. **Mucho mas!!!**

open(), close(), read(), write()

- Cuando escribimos a la pantalla estamos escribiendo a un archivo?
- Si!

```
#include <unistd.h>

int main() {
    write(1, "Hola, mundo\n", 12);
    return 0;
}
```

Ken Thompson (más viejo)

open(), close(), read(), write()

Por convención, los programas esperan que al empezar existan 3 File Descriptors estandar.

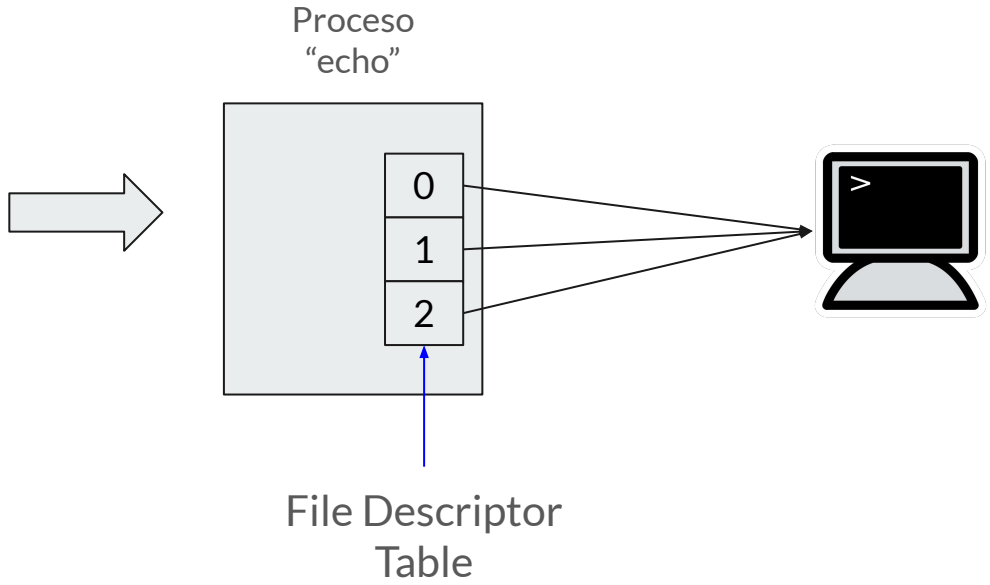
Table 4-1: Standard file descriptors

File descriptor	Purpose	POSIX name	*stdio* stream
0	standard input	STDIN_FILENO	*stdin*
1	standard output	STDOUT_FILENO	*stdout*
2	standard error	STDERR_FILENO	*stderr*

open(), close(), read(), write()

Quien hace el open de los FD estandar? El shell
Que cosa abre el shell? Por defecto, la terminal.

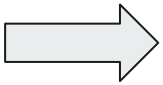
```
> echo “Hola mundo”
```

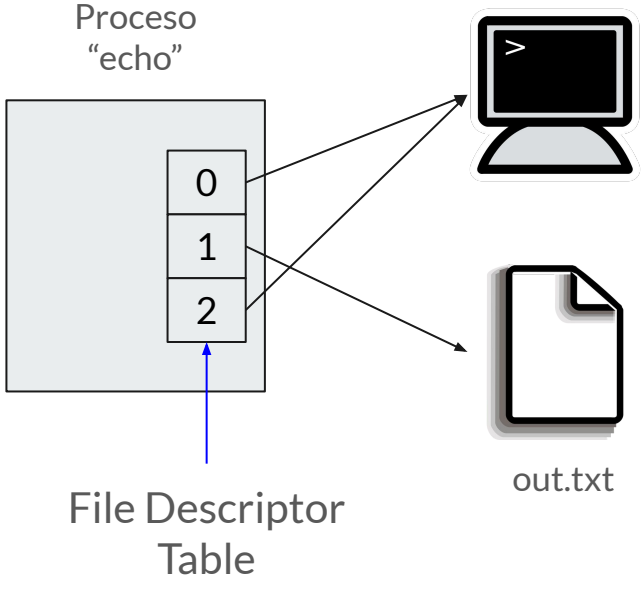


open(), close(), read(), write()

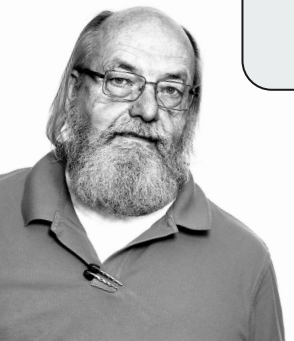
Pero el Shell puede hacer las cosas diferente!

```
> echo "Hola mundo" > out.txt
```





Echo no sabe que hay detrás del FD 1 realmente



Exactamente! Los FD **abstraen** la interfaz de lo que hay detrás!

Recapitulando

Cualquier cosa con “semántica de archivo” (que se comporta como un archivo) en Unix se puede escribir y leer como un archivo (ie. bytes)

Si tengo un File Descriptor, tengo un “archivo (en el sentido abstracto)”

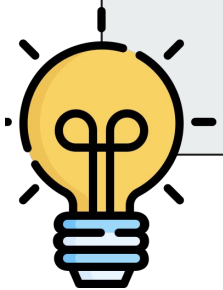
A veces los abro yo… a veces me los pasa mi proceso padre (eg. el shell)

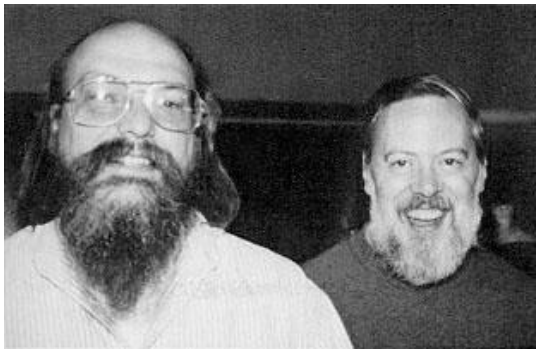
El “file” es la segunda abstracción más importante de Unix

(la primera es el “Proceso”)

Procesos

Un Proceso es una instancia de un programa en ejecución

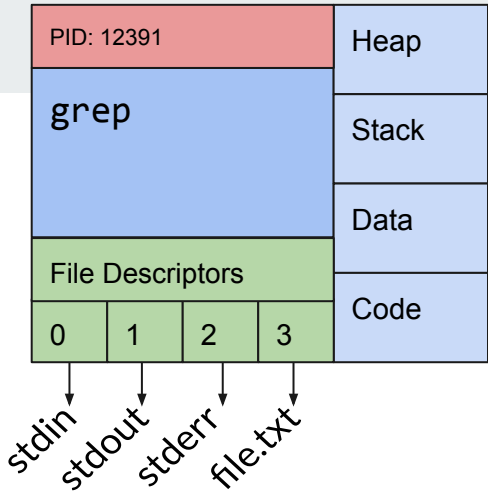




Ken Thompson y Dennis ritchie
Creadores de Unix

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

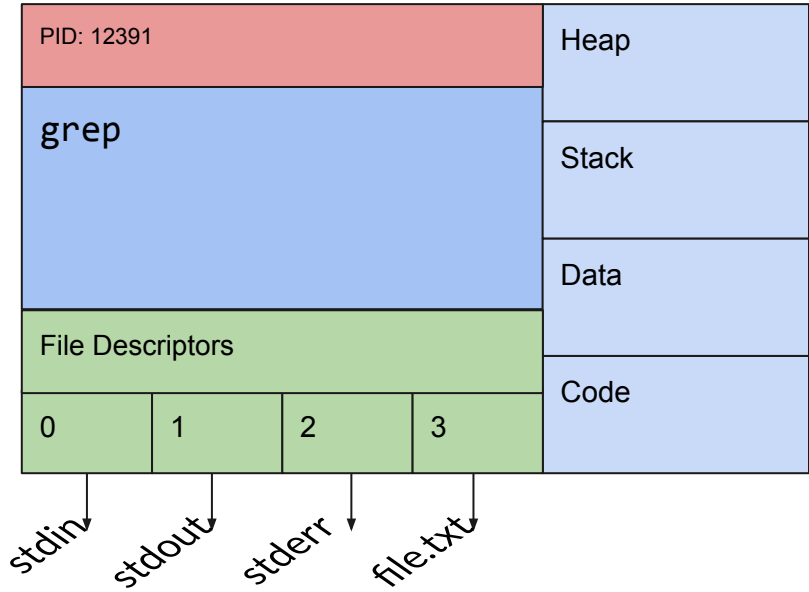
Partes básicas de un proceso:

PID:Process Id

Nombre del Programa

File Descriptors

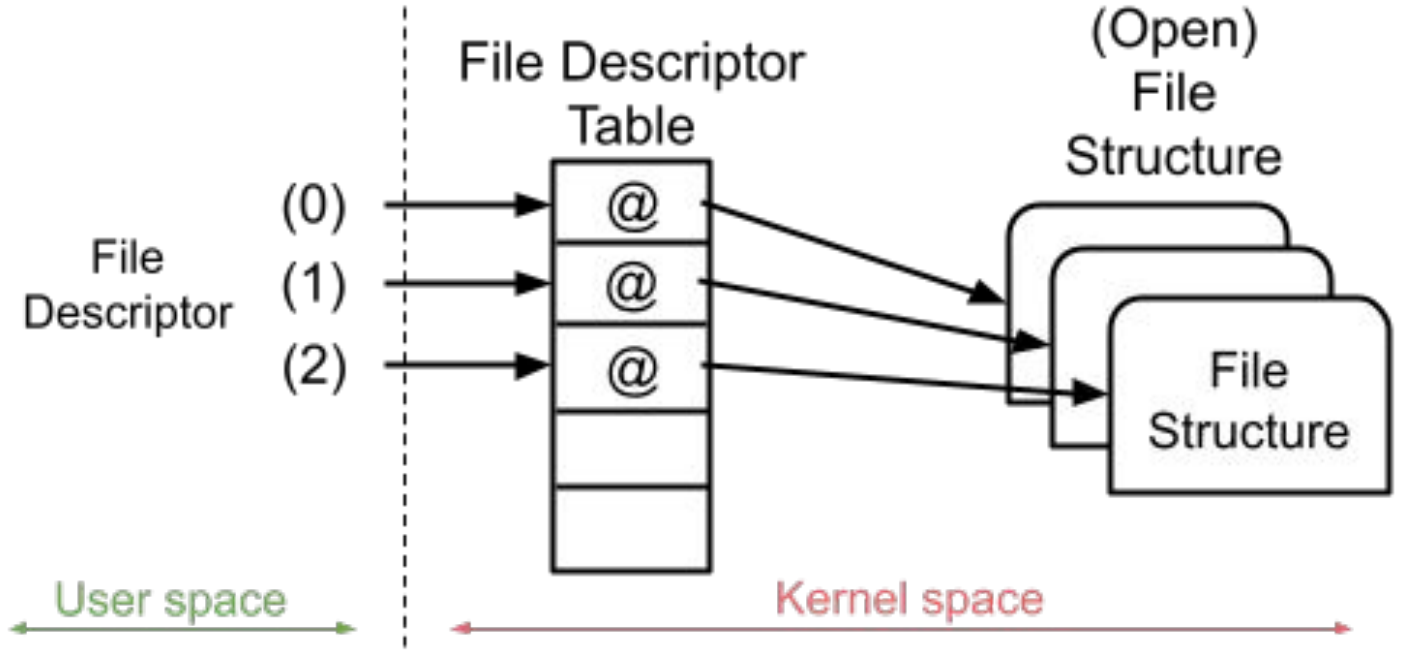
Memoria:codigo,datos,stack,heap



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

No se puede examinar la File Descriptor Table desde el proceso mismo. Es una estructura administrada internamente por el Sistema Operativo para cada proceso



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()

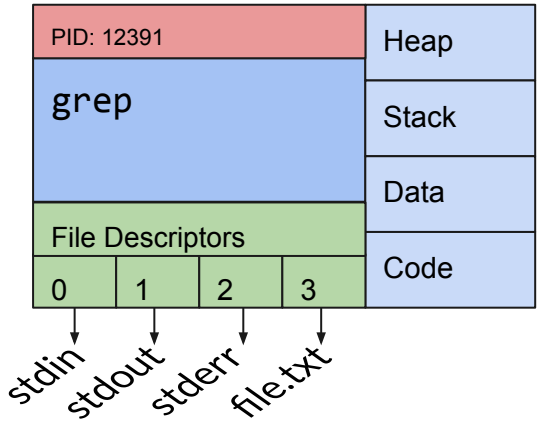
#include <unistd.h>
pid_t fork(void);

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.

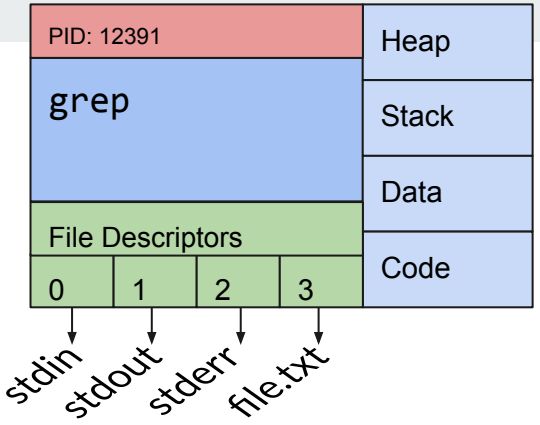
System Call fork()



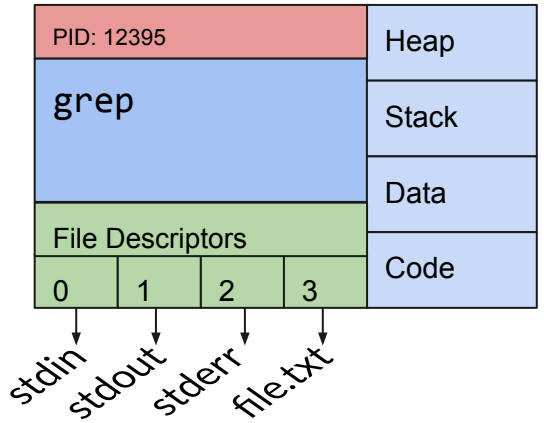
pid:12391

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 espacio de **memoria** del proceso padre.
- El hijo **hereda** los **File Descriptors** del proceso padre
- 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 heap y segmentos de datos (inicializados y no inicializados) recién inicializados.

System Call execve()

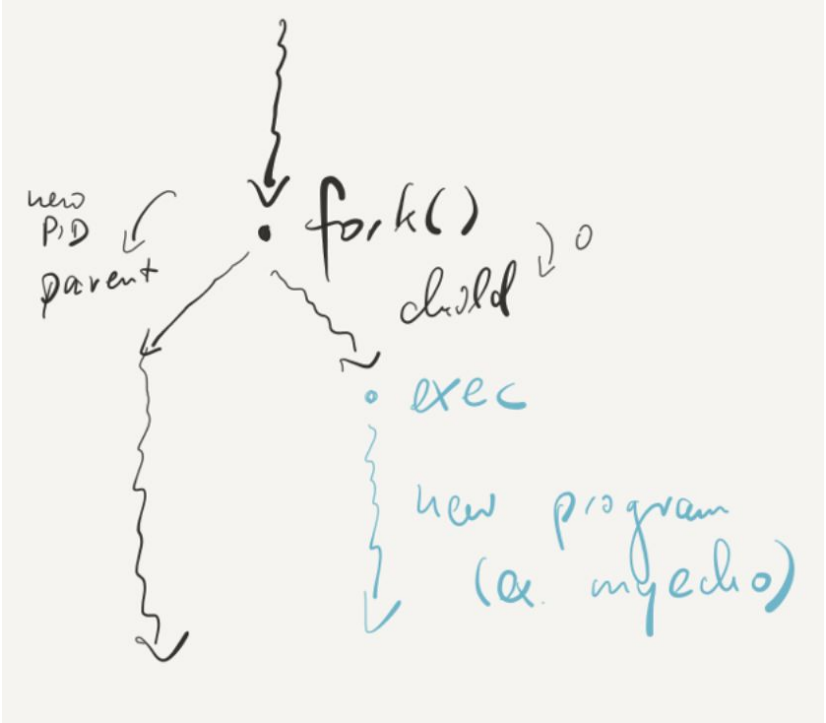
Ojo que no se crea ningún proceso solo se reemplaza
Un programa en ejecución por otro.

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");
      }
}
```

¿Que hace y que no hace execve?:

- Carga el programa y reemplaza el espacio de memoria del proceso por el del nuevo programa

- **No** crea un nuevo proceso.
- **No** cambia el pid
- **No** modifica los **File Descriptors** (el nuevo programa, ve la tabla como estaba antes del execve)

System Call dup()

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

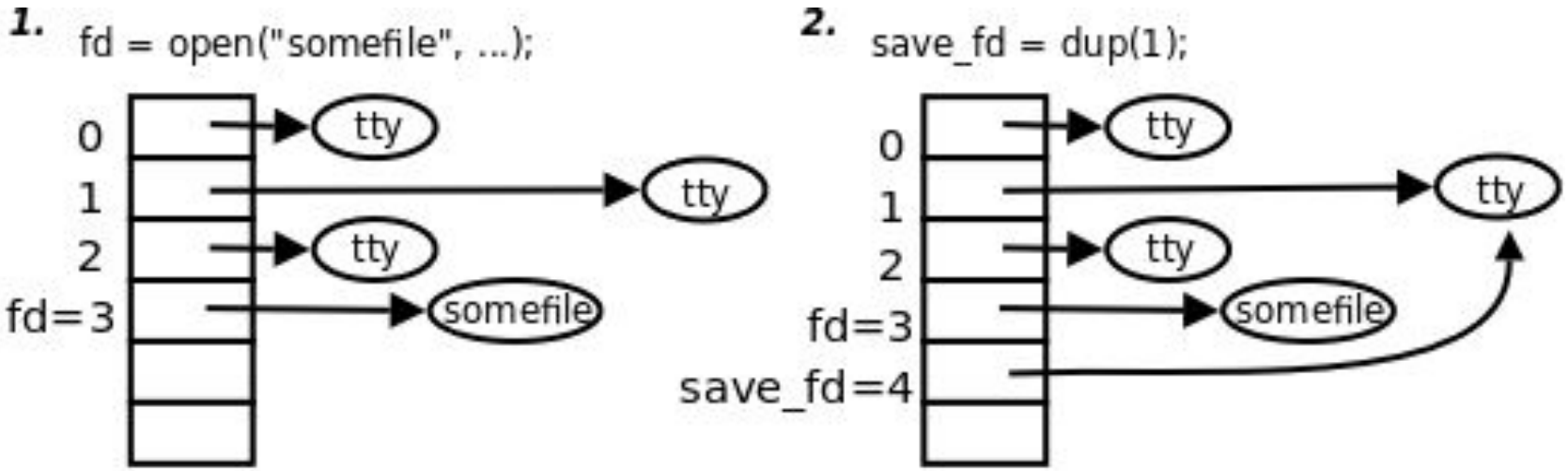
Estas System Calls duplican un file descriptor, el cual puede ser utilizado indiferentemente entre el y el duplicado.

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**.

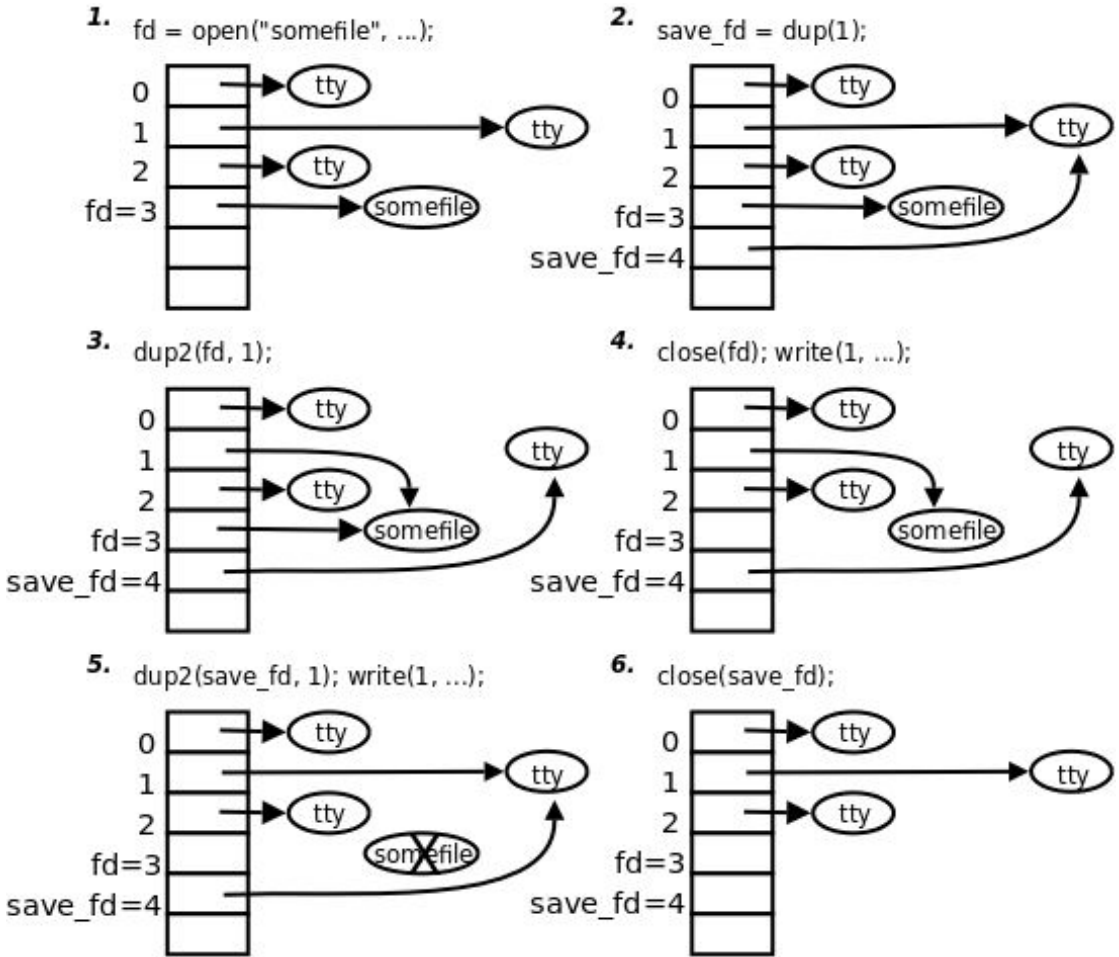
System Call dup()



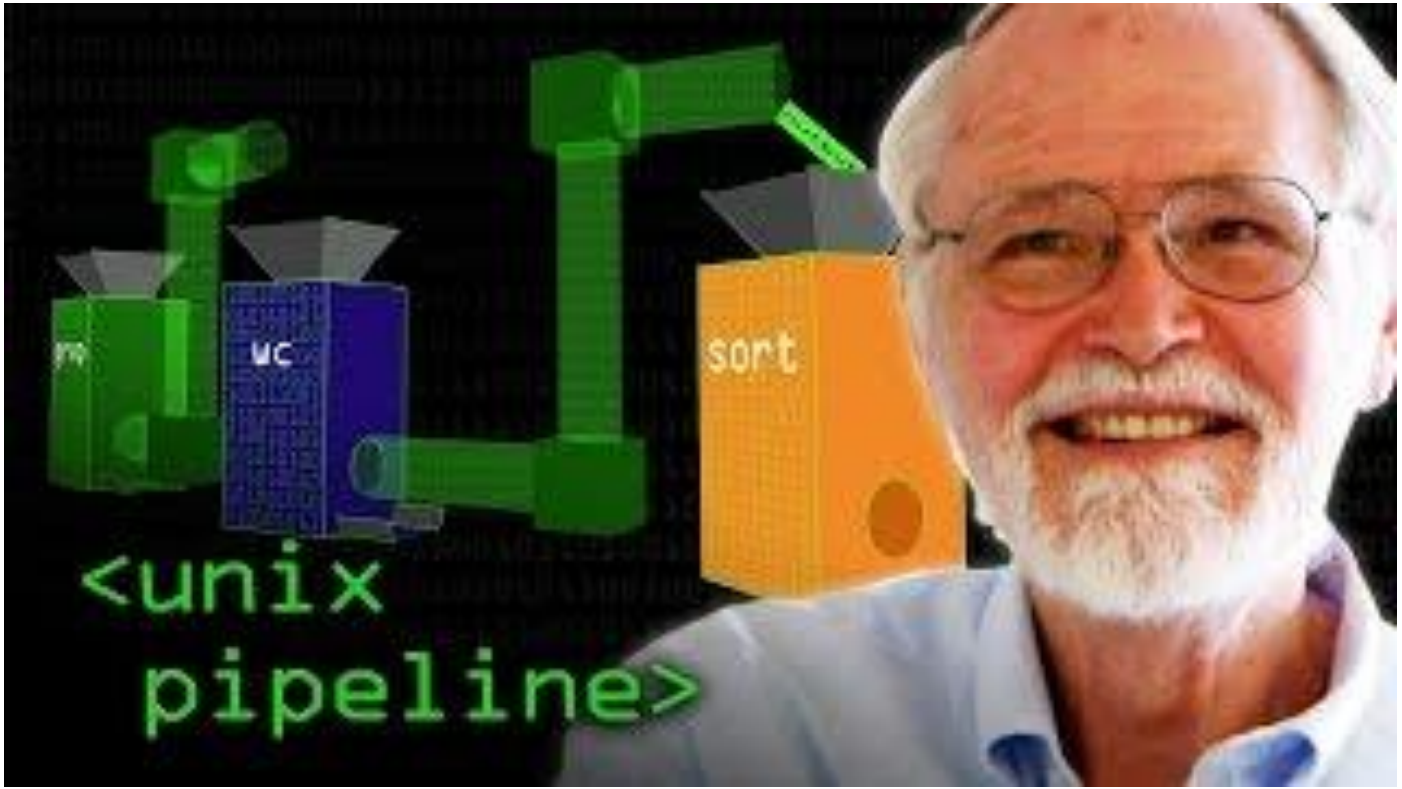
System Call dup()

```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);
}
```

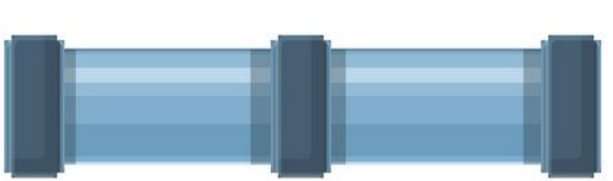
System Call dup()



Pipelines



System Call pipe()

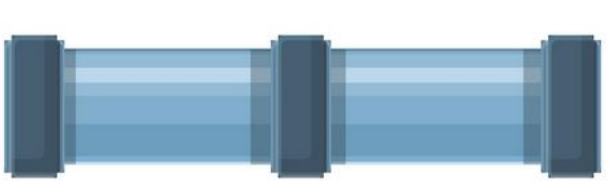


```
#include <unistd.h>
int pipe(int pipefd[2]);
```

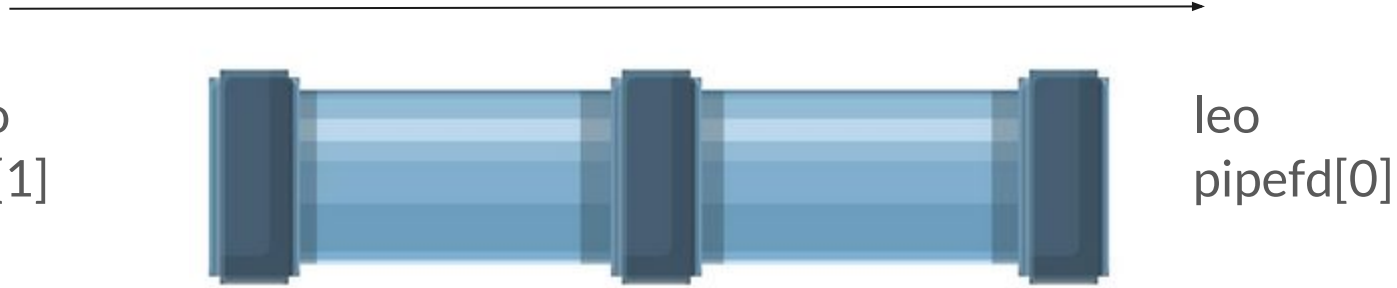
Un pipe es un **pequeño buffer en el kernel** expuesto a los procesos como un par de files descriptors, uno para escritura y otro para lectura. Al escribir datos en el extremo el pipe hace que estos estén disponibles en el otro extremo para ser leídos.

ES UN CANAL UNIDIRECCIONAL!!!

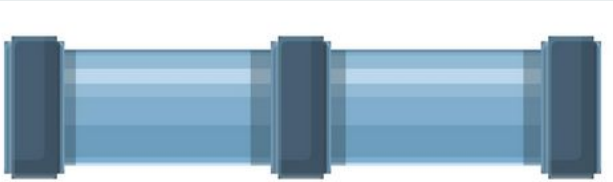
System Call pipe()



Int pipefd[2]

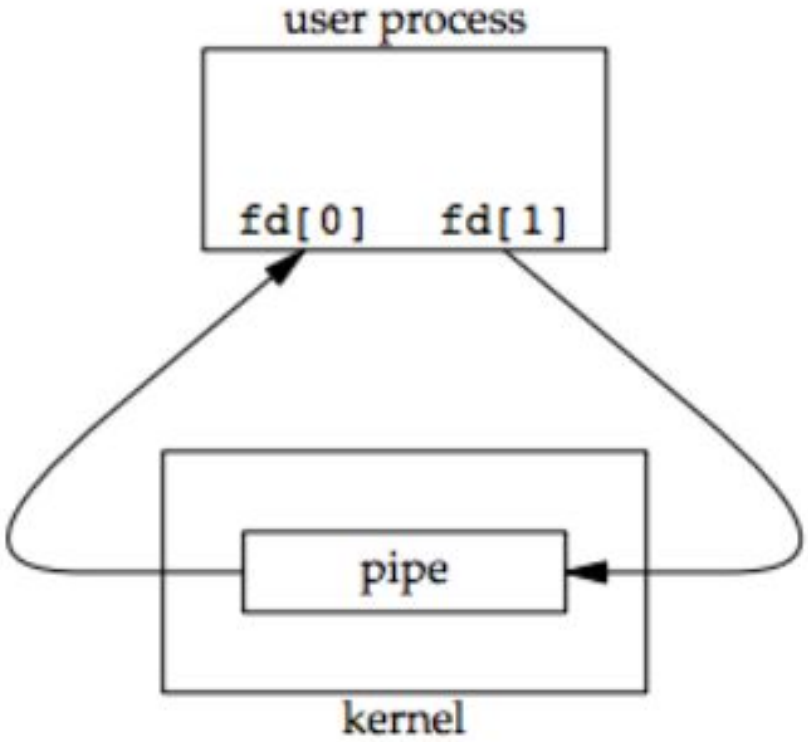


System Call pipe()

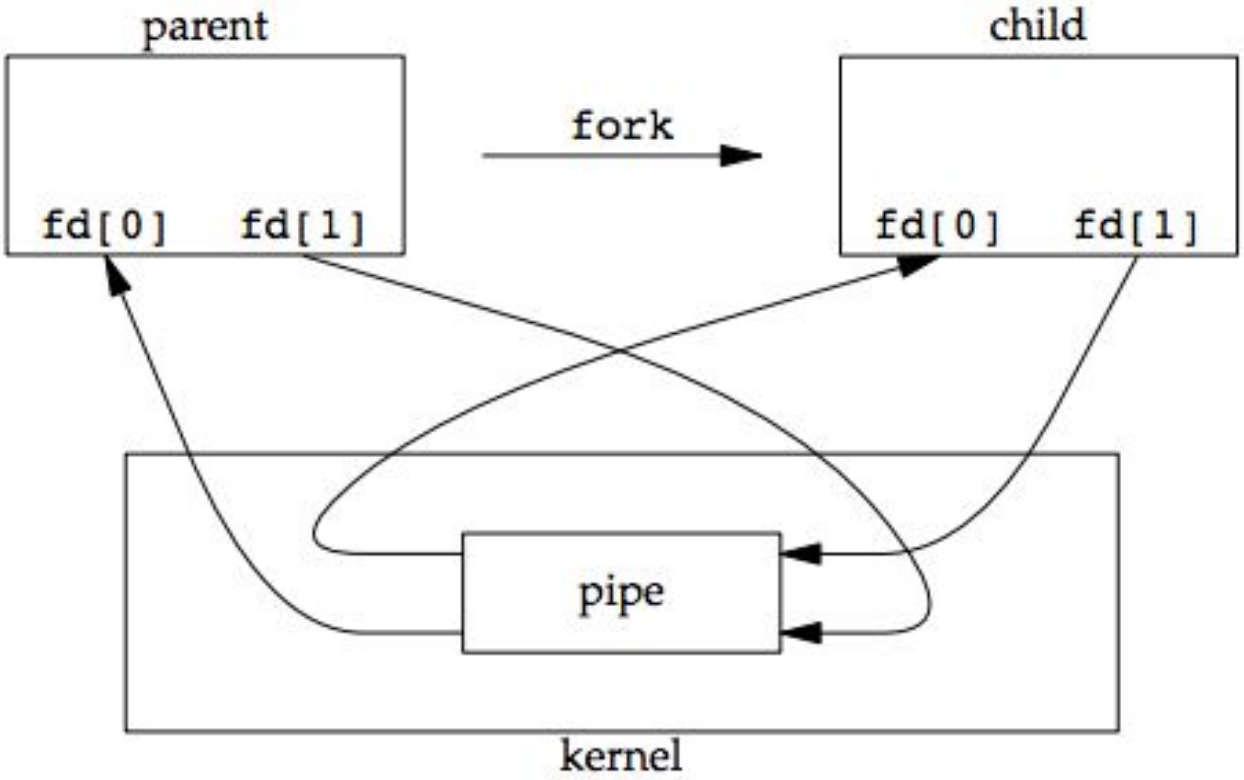


Puedo hacer que dos procesos compartan información.

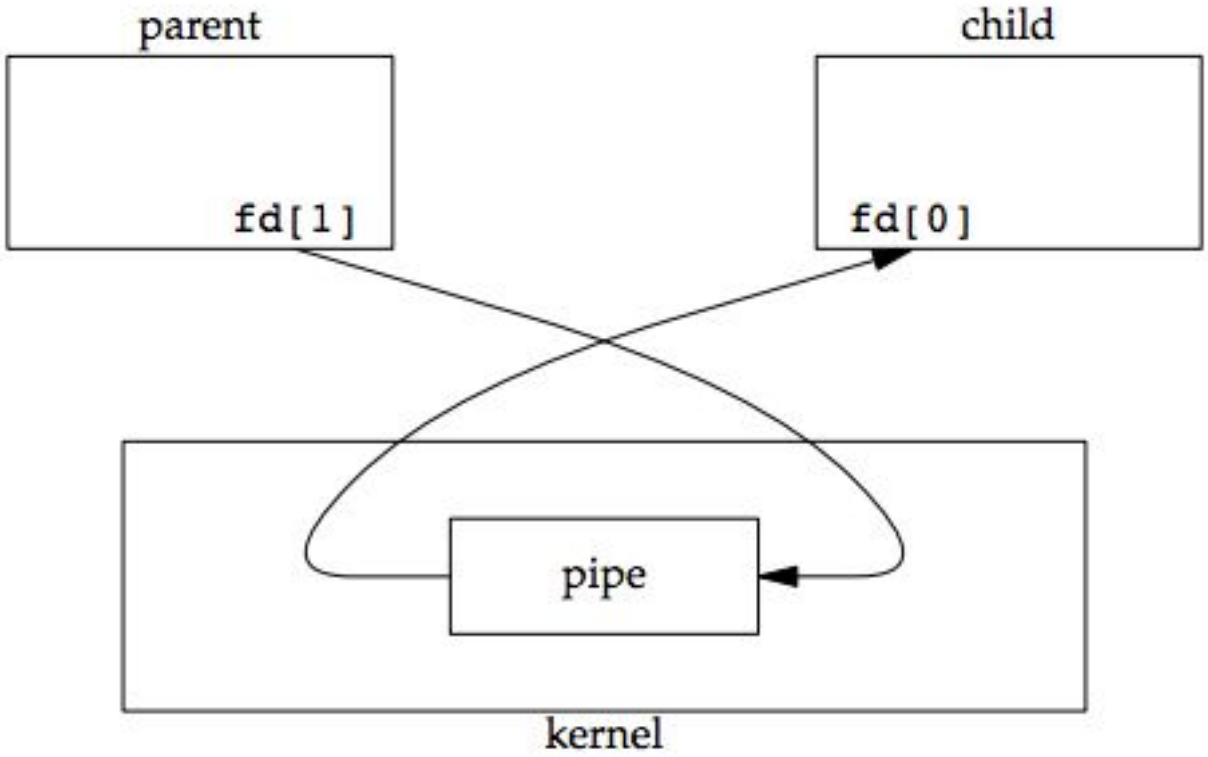
System Call pipe()



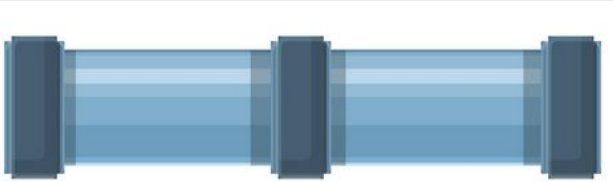
System Call pipe()



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]);
}
```