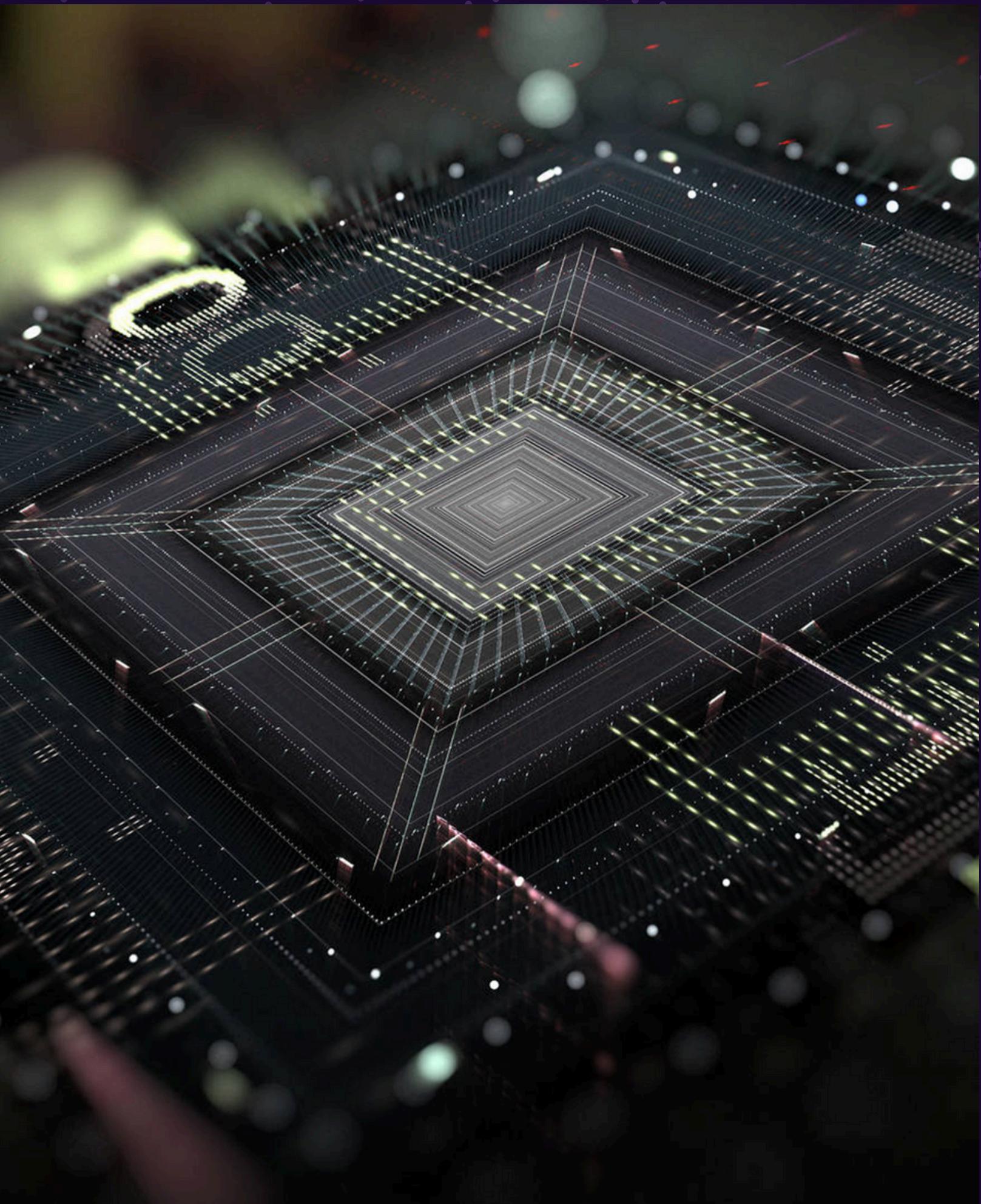


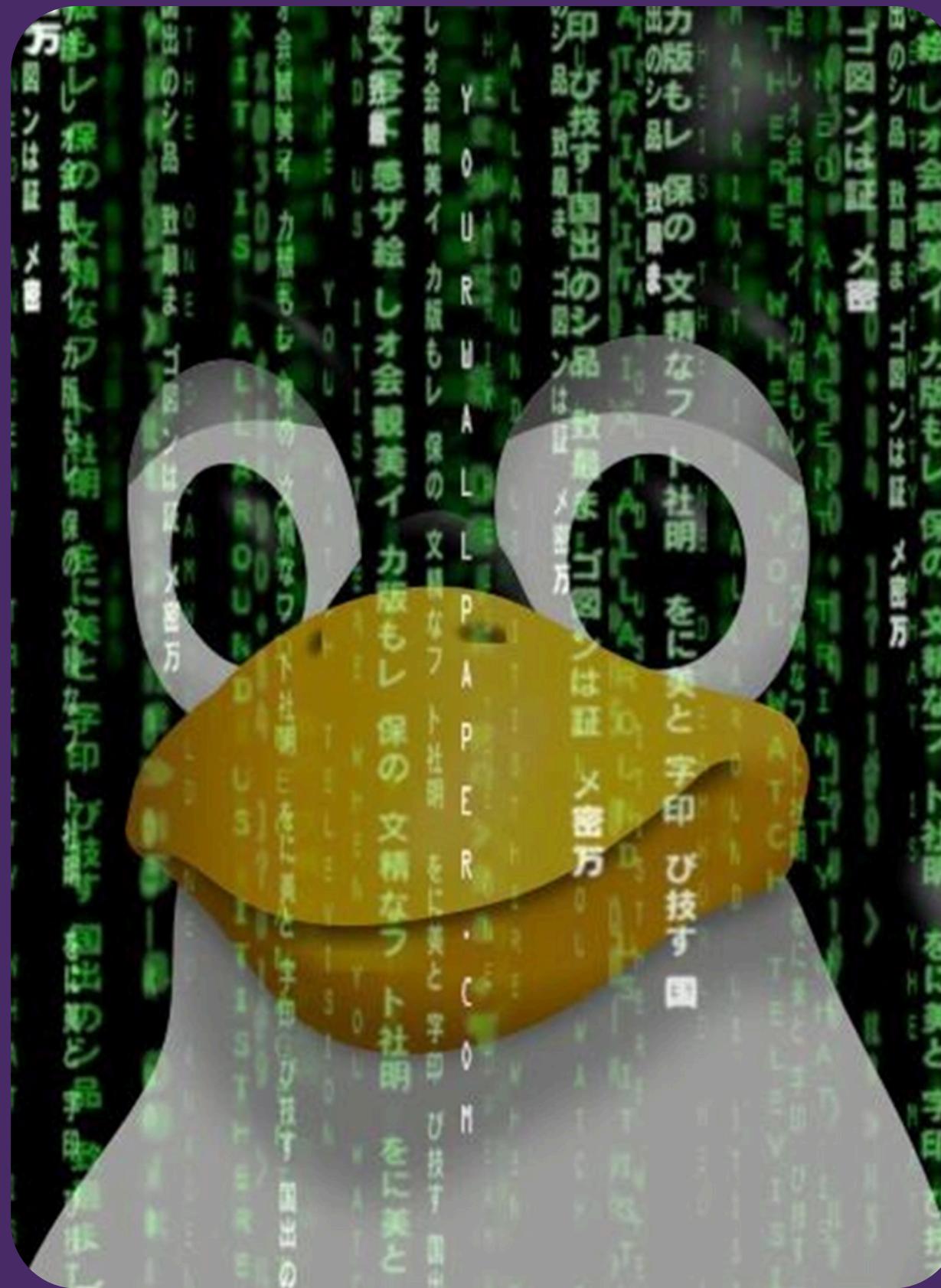
LABORATORIO SISTEMAS OPERATIVOS 2

LLAMADAS AL SISTEMA

Los sistemas operativos deben ofrecer a los usuarios no sólo el mayor nivel de comodidad posible, sino también la máxima estabilidad y seguridad, por lo que los desarrolladores de sistemas se esfuerzan por mantener lo más bajo posible el riesgo de posibles complicaciones del sistema debido a negligencias involuntarias o ataques dirigidos desde el exterior.

Uno de los pasos más importantes adoptados con este fin es la separación estricta del núcleo del sistema operativo (el kernel) y los programas de aplicación o procesos de usuario, es por esto que programas y procesos que no pertenecen al sistema no tienen acceso directo a la CPU ni a la memoria, sino que dependen de las llamadas llamadas al sistema.





LLAMADAS AL SISTEMA

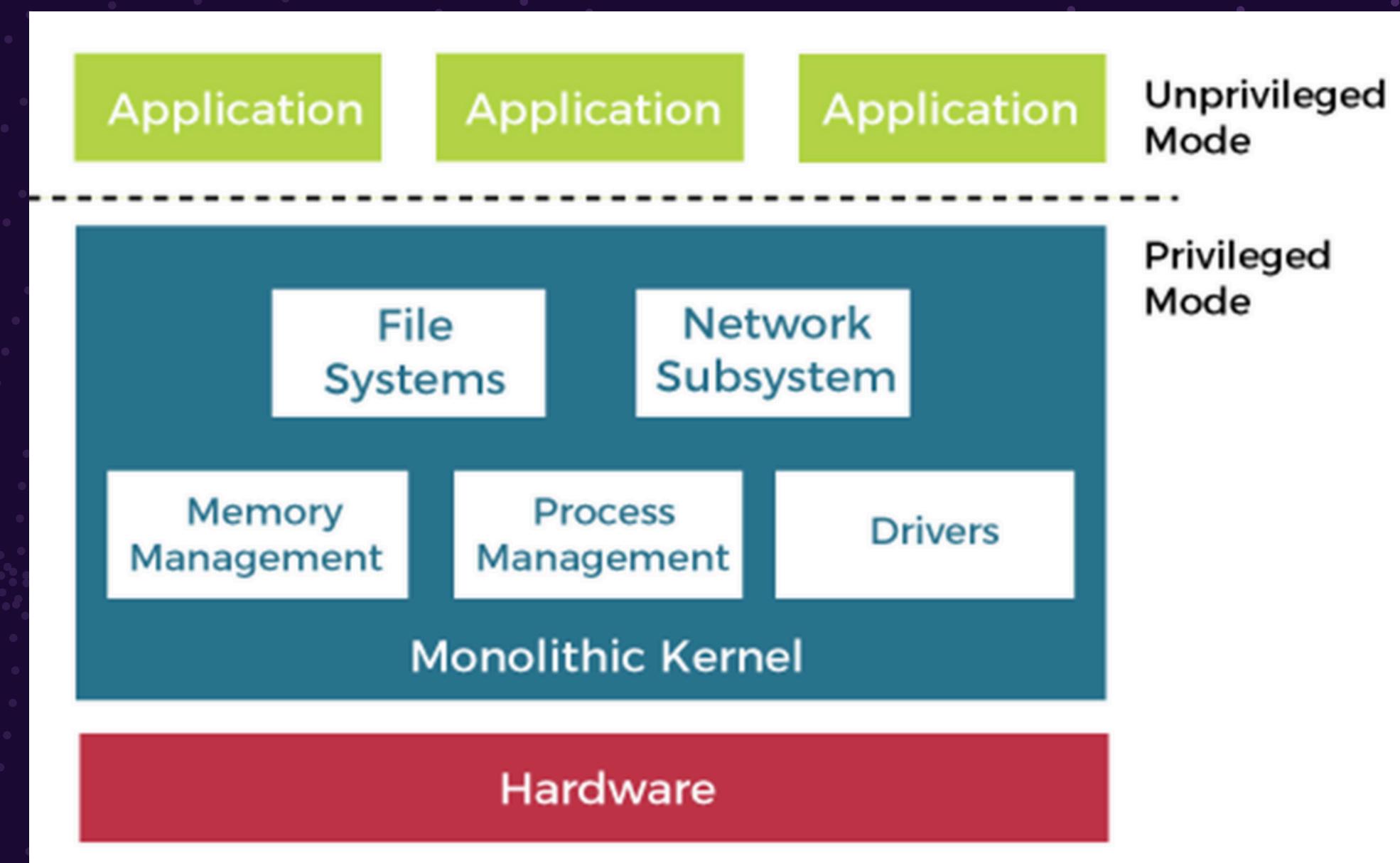
Una llamada al sistema, o syscall, es una rutina que permite a una aplicación de usuario solicitar acciones que requieren privilegios especiales. La adición de llamadas al sistema es una de varias maneras de ampliar las funciones proporcionadas por el kernel.

En los sistemas operativos modernos, este método se utiliza si una aplicación o proceso de usuario necesita pasar información al hardware, a otros procesos o al propio kernel, o si necesita leer información de estas fuentes.

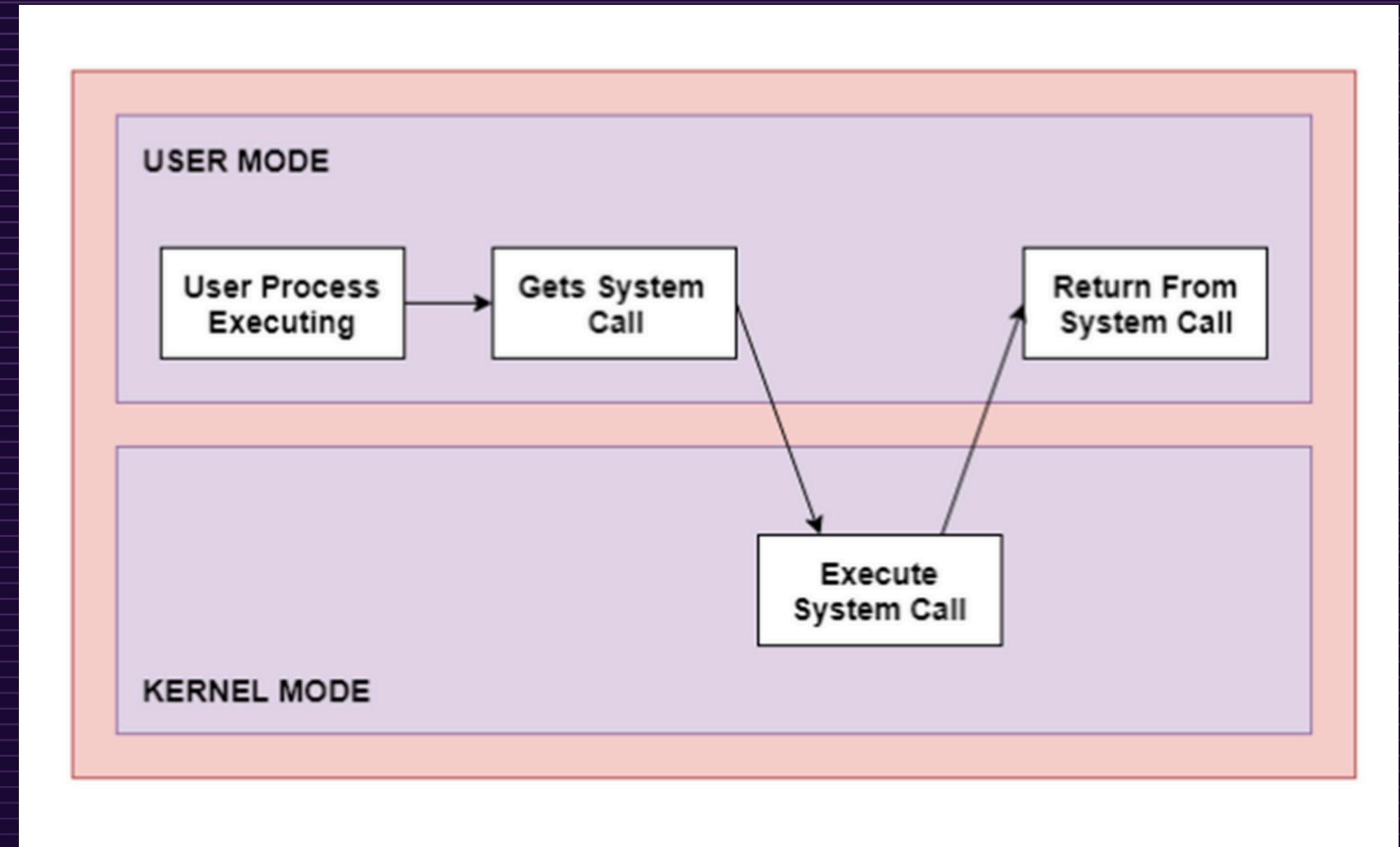
Esto hace que estas llamadas sean un vínculo entre el modo de usuario y el modo kernel

La necesidad de **llamadas al sistema** está estrechamente ligada al modelo de sistema operativo moderno con modo de usuario y modo kernel (monolítico).

Procesos en **modo usuario** se utilizan normalmente para realizar cálculos, acceder a recursos a nivel de usuario, como archivos y memoria, y gestionar el control de procesos. Estas tienen acceso limitado a los recursos del sistema y garantiza que los procesos no puedan interferir entre sí.



El **modo kernel** es el sistema de control fundamental donde se ejecutan todos los servicios y procesos del sistema, así como realizar las acciones críticas del sistema por parte de los programas de aplicación que están bloqueados en el modo de usuario, como configurar asignaciones de memoria o acceder a dispositivos de E/S.

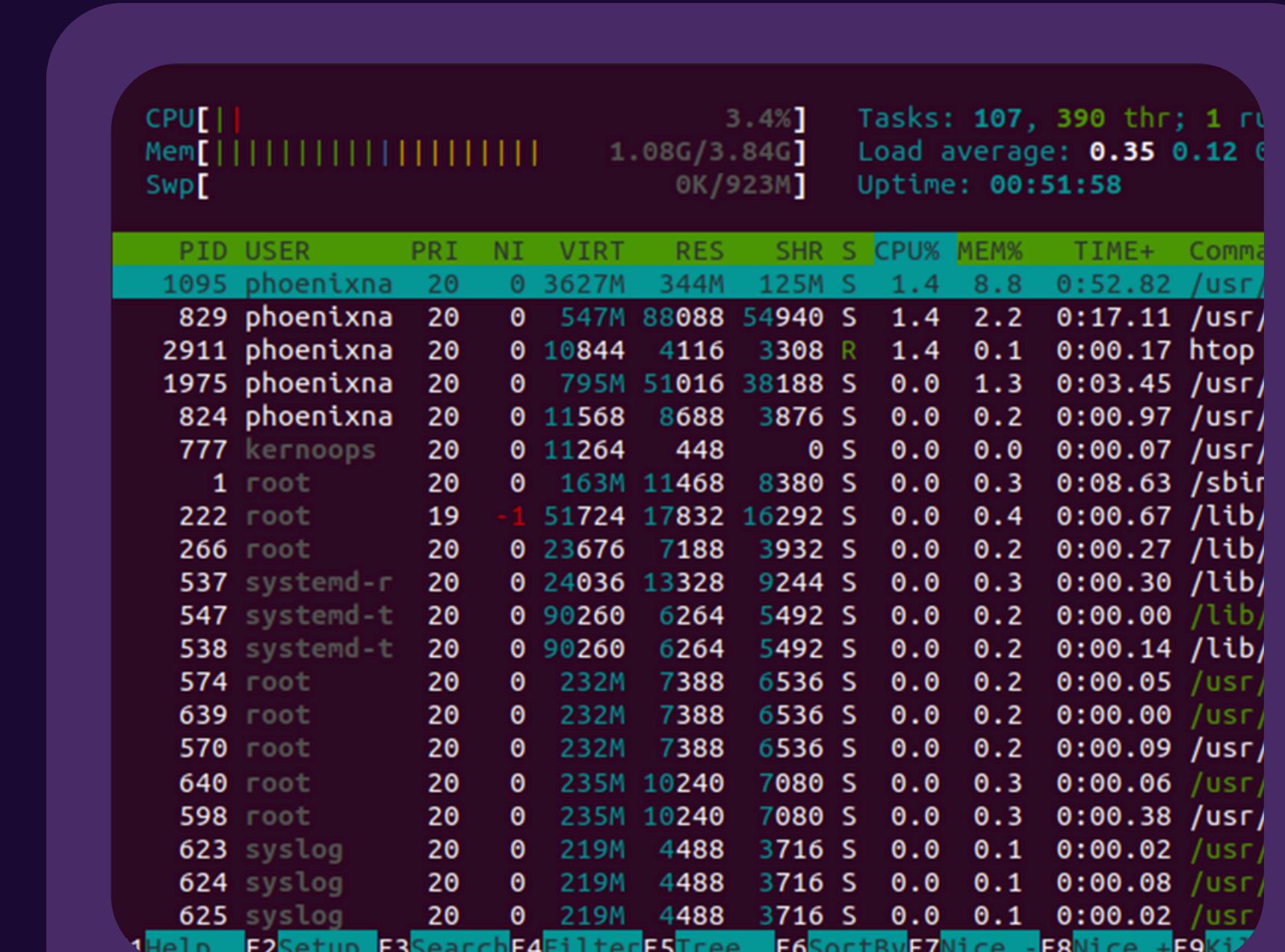


En Linux, realizar una llamada al sistema implica transferir el control del modo de usuario sin privilegios al modo de kernel privilegiado; Los detalles de esta transferencia varían de una arquitectura a otra. Las bibliotecas se encargan de recopilar los argumentos de la llamada al sistema y, si es necesario, de organizarlos en la forma especial necesaria para realizar la llamada al sistema.

LLAMADAS AL SISTEMA PARA GESTIÓN DE PROCESOS

La interfaz de llamadas de usuario suministra al desarrollador de software herramientas tanto para la creación, sincronización y comunicación de nuevos procesos, como la capacidad de ejecutar nuevos programas.

Las llamadas al sistema para gestión de procesos de Linux son `fork()`, `exit()`, `exec()`, aunque también exploraremos las señales entre procesos, utilizadas para la comunicación entre procesos.



FORK()



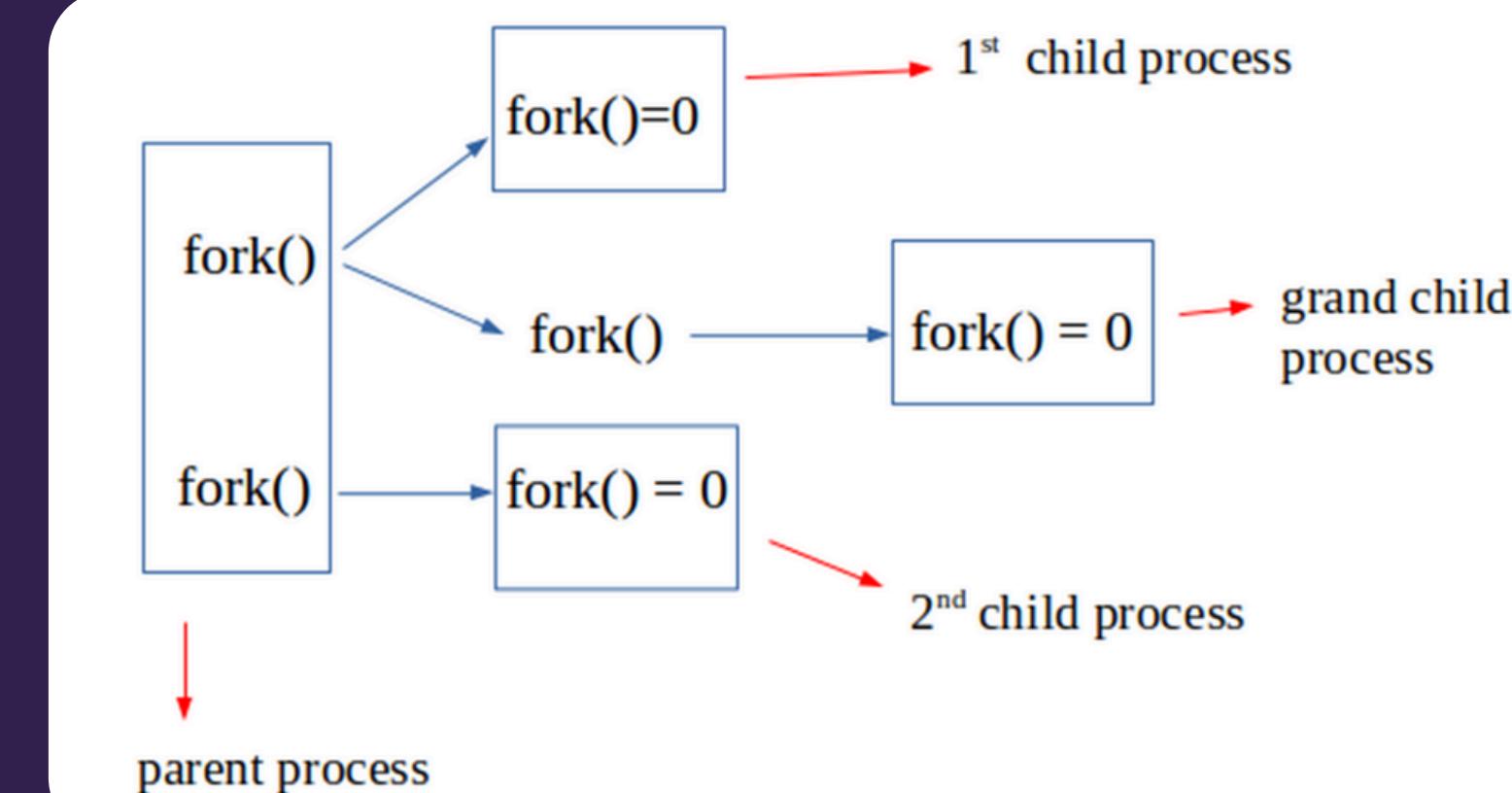
El kernel crea un nuevo proceso (proceso hijo) realizando una copia del proceso que realiza la llamada al sistema fork (proceso padre). Así, salvo el PID y el PPID los dos procesos serán inicialmente idénticos. De esta forma los nuevos procesos obtienen una copia de los recursos del padre (heredan el entorno).

El proceso hijo tendrá copias de los descriptores, si los utiliza el proceso padre. Sin embargo, las copias de los descriptores harán referencia a los mismos objetos subyacentes.

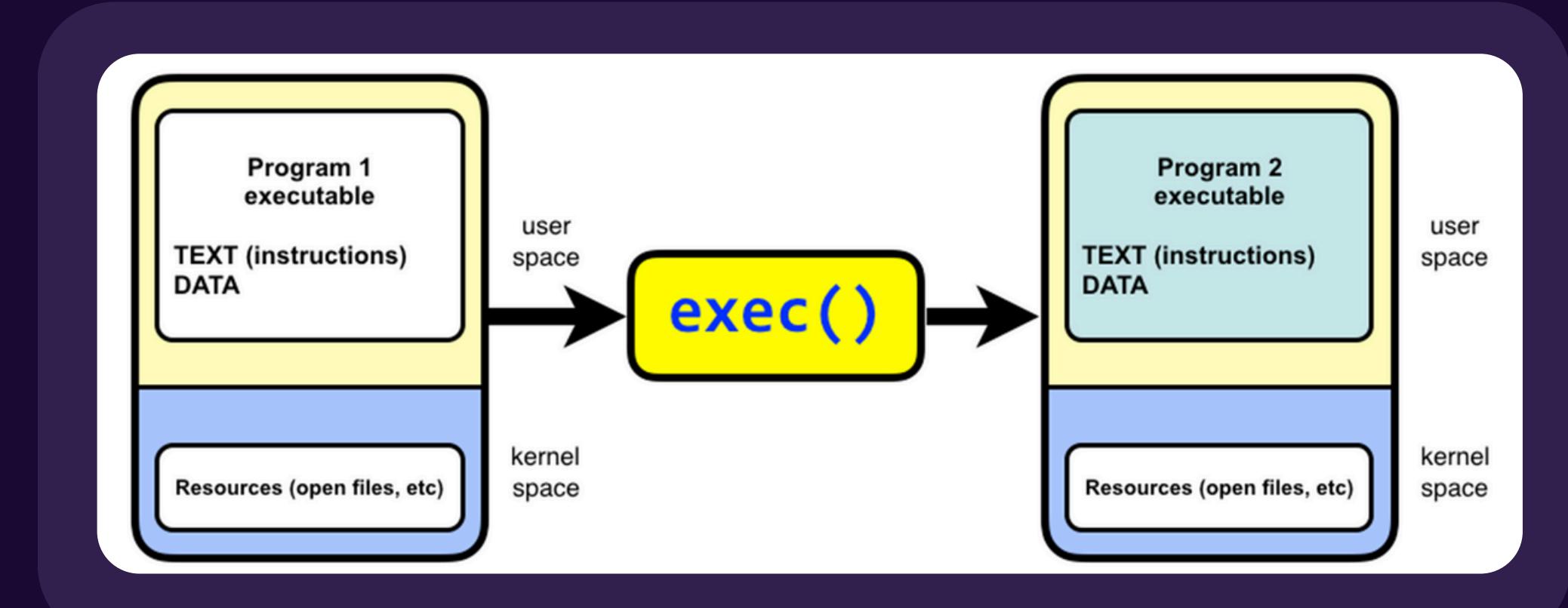
Su sintaxis es:
int pid = fork ();

Este retornará un valor diferente dependiendo de donde sea ejecutado, sus posibles valores son:

- 0 cuando se ejecuta desde el proceso hijo
- PID del nuevo proceso cuando se ejecuta del proceso padre
- -1 si ocurre un error al crear el nuevo proceso



EXEC()

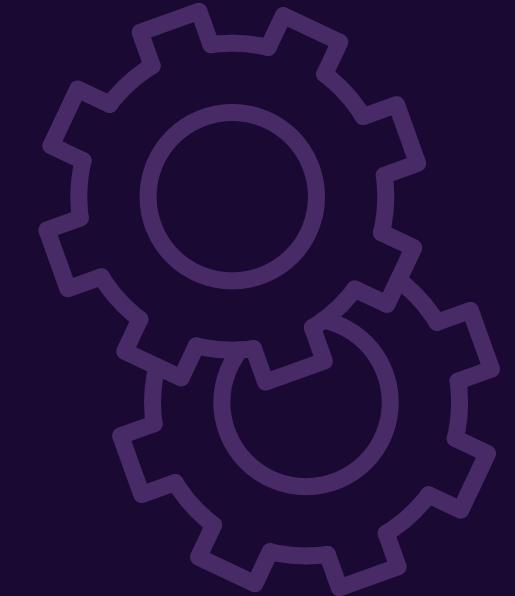


La llamada al sistema exec permite remplazar los segmentos de instrucciones y de datos de usuario por otros nuevos a partir de un archivo ejecutable en disco, con lo que se consigue que un proceso deje de ejecutar instrucciones de un programa y comience a ejecutar instrucciones de un nuevo programa. exec() no crea ningún proceso nuevo.

El programa que se está ejecutando actualmente finaliza inmediatamente y el nuevo programa comienza a ejecutarse en el contexto del proceso existente.

Hay 6 formas de realizar una llamada al sistema exec:

- int execl (**char *path**, char *arg0, char *arg1, . . . , char *argN, char *null)
- int execlp (**char *file**, char *arg0, char *arg1, . . . , char *argN, char *null)
- int execle (**char *path**, char *arg0, char *arg1, . . . , char *argN, char *null, char *envp[])
- int execv (**char *path**, char *argv[])
- int execvp (**char *file**, char *argv[])
- int execve (**char *path**, char *argv[], char *envp[])



Donde:

- **path** = ruta del ejecutable del nuevo programa a ejecutar.
- **file** = nombre del nuevo programa a ejecutar.
- **arg0** = primer argumento del programa. Por convención suele asignarse el nombre del programa sin la trayectoria.
- **arg1 ... argN** = Conjunto de parámetros que recibe el programa para su ejecución.
- **argv** = Matriz de punteros a cadenas de caracteres. Estas cadenas de caracteres constituyen la lista de argumentos disponibles para el nuevo programa.
- **envp** = Matriz de punteros a cadenas de caracteres. Estas cadenas de caracteres constituyen el entorno de ejecución del nuevo programa.

WAIT()



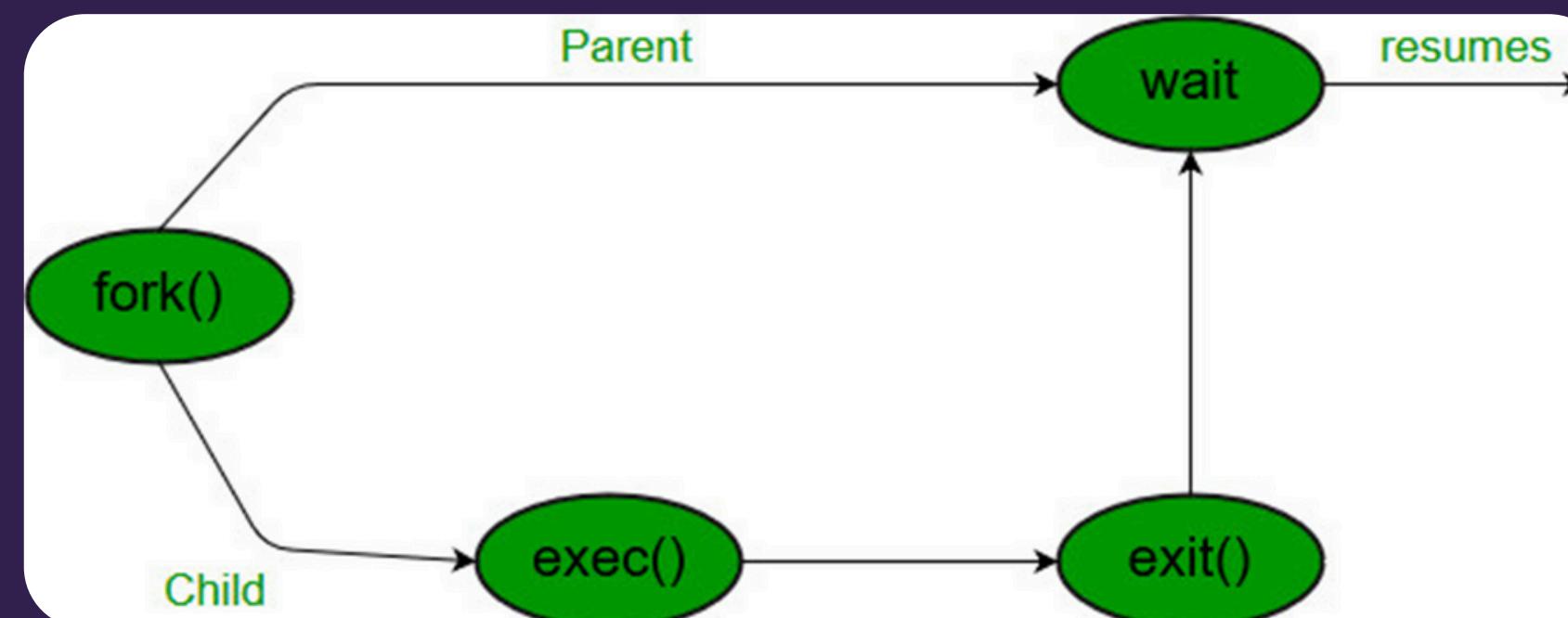
La llamada al sistema wait() suspende la ejecución del hilo que lo llama hasta que uno de sus hijos termine.

Se utiliza un parámetro 'status' para comunicar al proceso padre la forma en que el proceso hijo termina. Por convenio, este valor suele ser 0 si el proceso termina correctamente y cualquier otro valor en caso de terminación anormal.

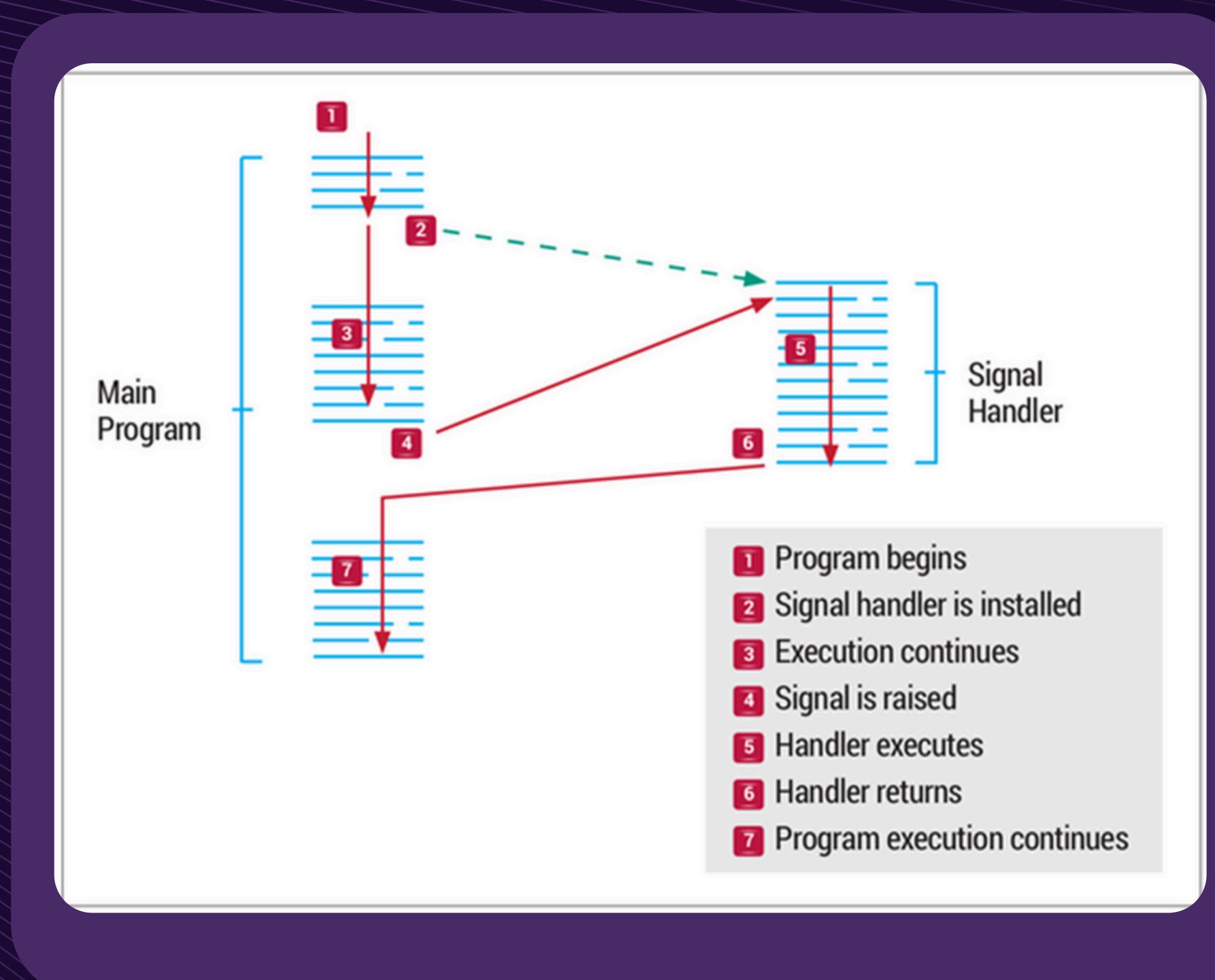
EXIT()



Un programa utiliza la llamada al sistema exit() para finalizar su ejecución. El sistema operativo recupera recursos que fueron utilizados por el proceso después de la llamada al sistema exit().



SEÑALES ENTRE PROCESOS



Se define una señal, como un mensaje enviado a un proceso determinado. Este mensaje no es más que un número entero.

Un proceso cuando recibe una señal puede optar por tres posibles alternativas para procesarla:

- Ignorar la señal recibida.
- Ejecutar una acción por defecto.
- Ejecutar una acción determinada, especificada por el usuario en el propio proceso.

SEÑALES MAS COMUNES



SIGINT

Señal de interrupción:
Enviada desde el terminal
(teclado) por medio de
Ctrl+c.



SIGQUIT

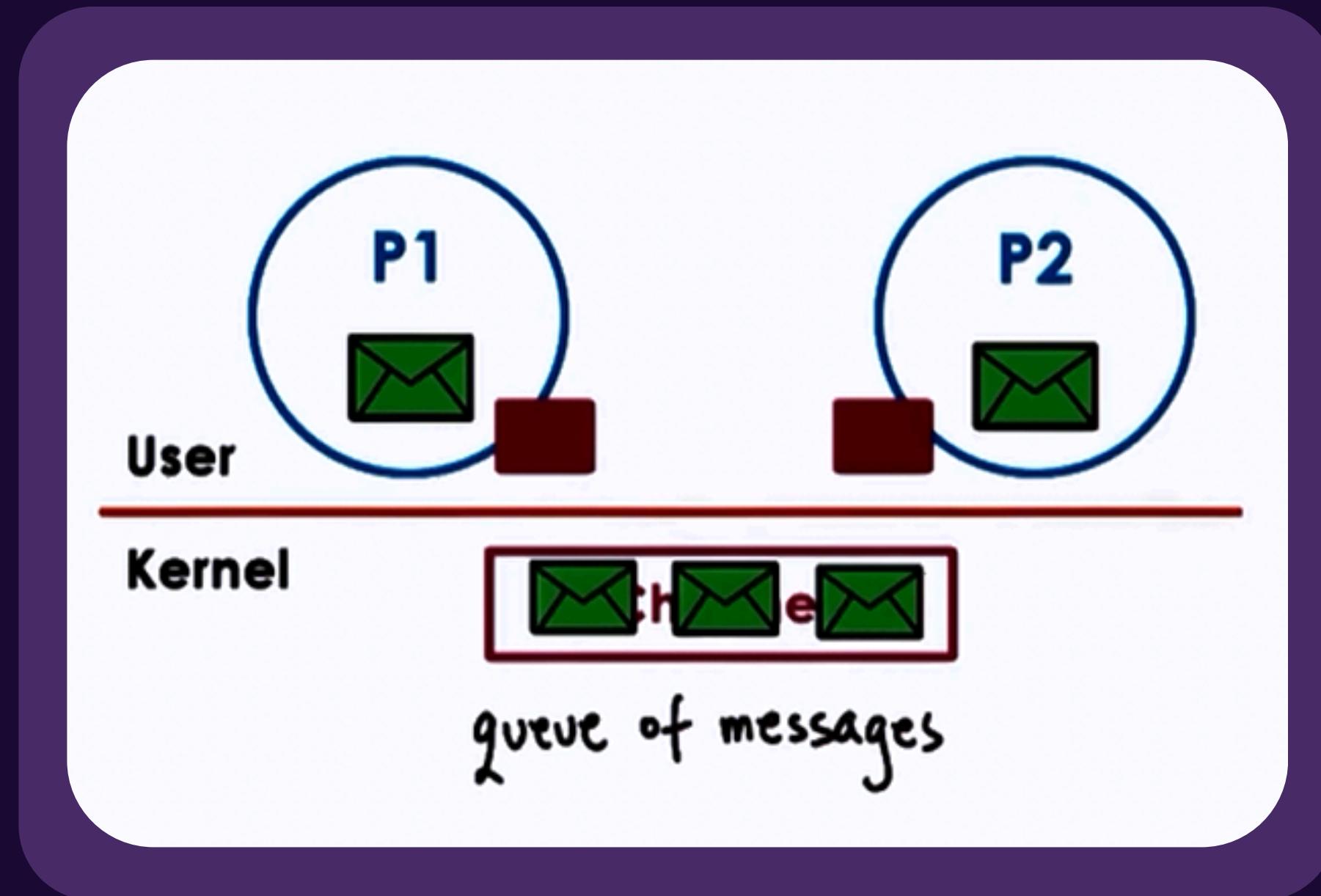
Similar a SIGINT, excepto
que produce un core dump
cuando finaliza el proceso,
como una señal de error de
programa. Puede considerar
esto como una condición de
error del programa
"detectada" por el usuario.



SIGKILL

Se utiliza para provocar la
terminación inmediata del
programa. No se puede
manejar ni ignorar y, por lo
tanto, siempre es fatal.
Tampoco es posible
bloquear esta señal.

LLAMADAS DE SISTEMA PARA COMUNICACIÓN ENTRE PROCESOS

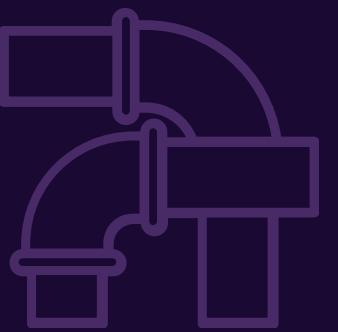


La comunicación entre procesos (Inter-Process Communication o IPC) es un mecanismo que permite que los procesos se comuniquen entre sí y sincronicen sus acciones. La comunicación entre procesos puede verse como un método de cooperación entre ellos.

La comunicación puede ser de dos tipos:

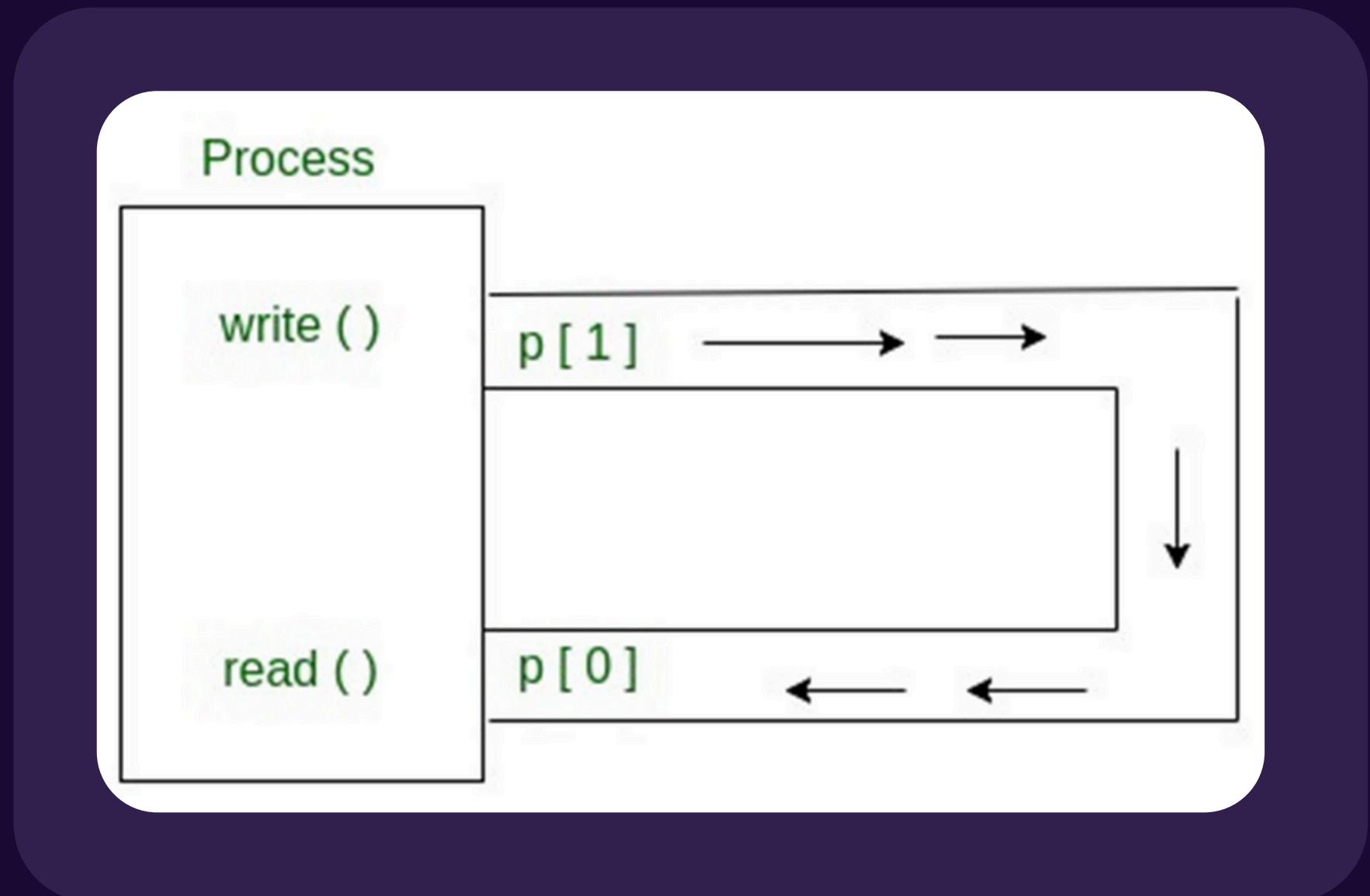
- Entre procesos relacionados que se inician desde un solo proceso, como procesos padre e hijo.
- Entre procesos no relacionados, o dos o más procesos diferentes.

PIPE()



Conceptualmente, pipe es una conexión entre dos procesos, de modo que la salida estándar (stdout) de un proceso se convierte en la entrada estándar (stdin) del otro proceso.

Es una comunicación unidireccional, es decir, podemos usar una pipe de modo que un proceso escriba en ella y otro proceso lea desde ella, haciendo que un área de la memoria principal sea tratada como un "archivo virtual".

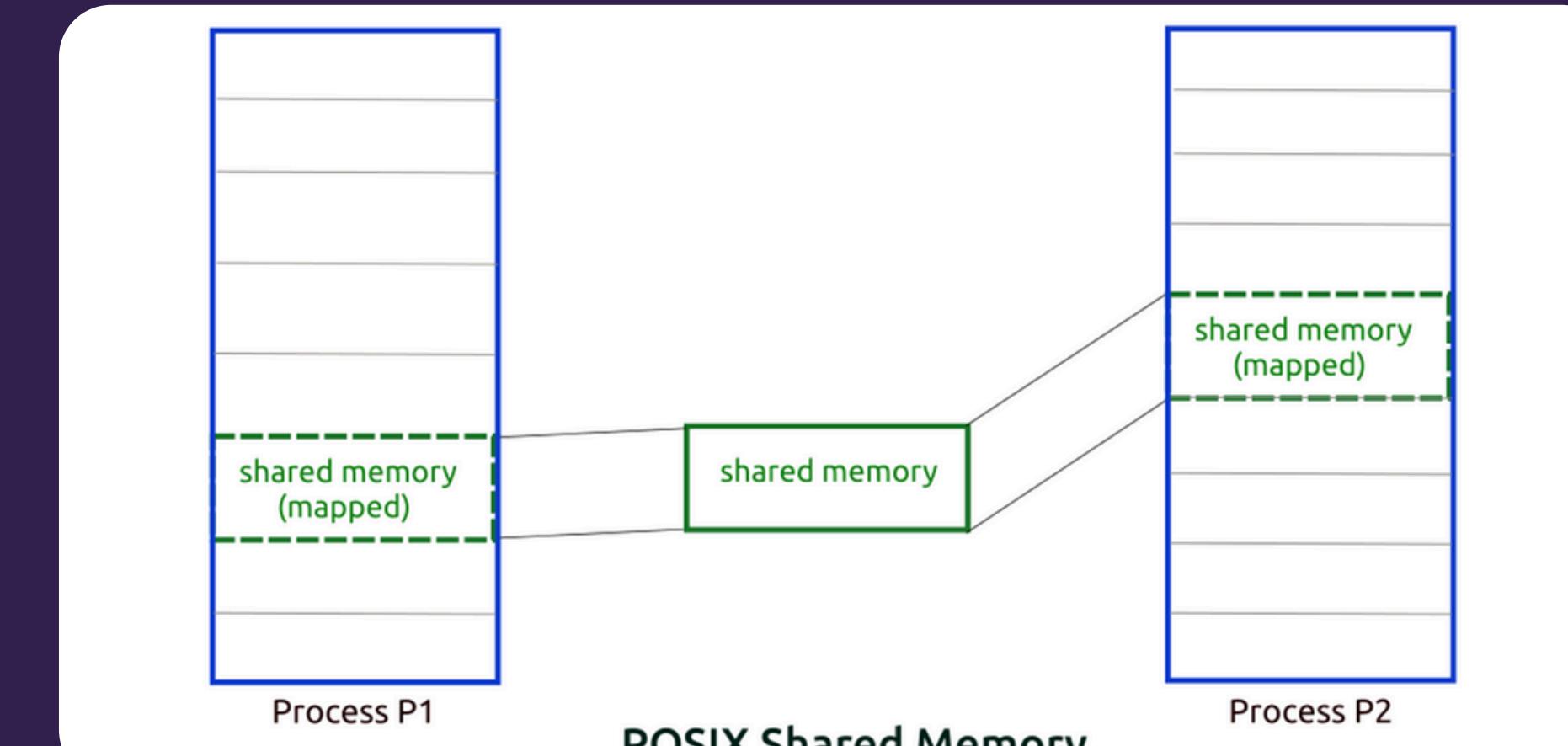


SHMGET()



Es una llamada al sistema que se utiliza para obtener un segmento de memoria compartida.

La memoria compartida es un mecanismo que permite que múltiples procesos compartan una región de memoria, lo cual es particularmente útil para la comunicación entre procesos (IPC).



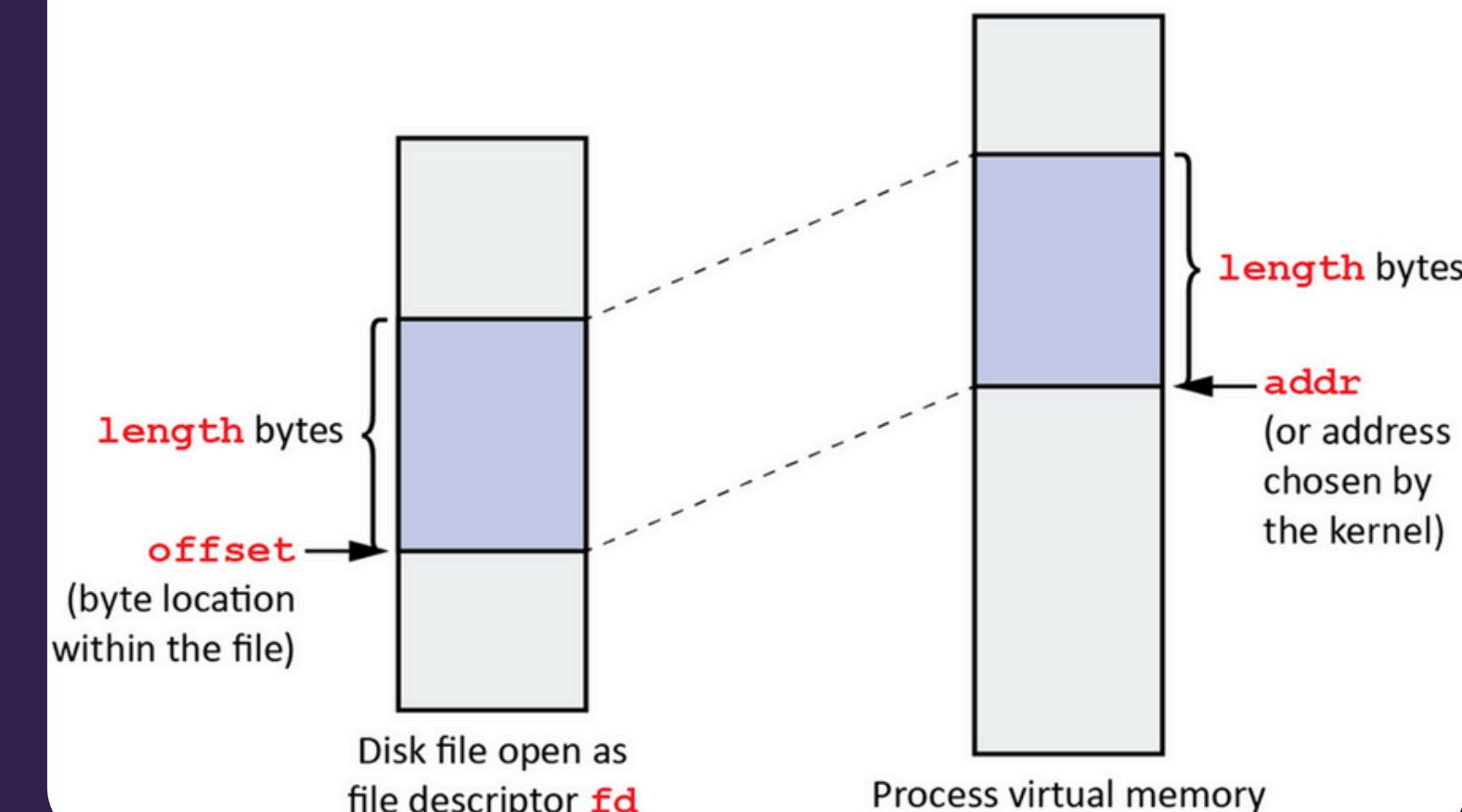
MMAP()



Es una llamada que proporciona una forma para que los procesos asigne archivos o dispositivos a la memoria.

Permite establecer un mapeo entre una región del espacio de direcciones virtuales del proceso y un objeto externo como un archivo, un dispositivo o un segmento de memoria anónimo. Este mapeo permite que el proceso lea o escriba en la región de la memoria como si fuera una matriz, y los cambios podrían afectar el objeto subyacente.

```
void *mmap(void *addr, size_t length,  
           int prot, int flags, int fd, off_t offset)
```

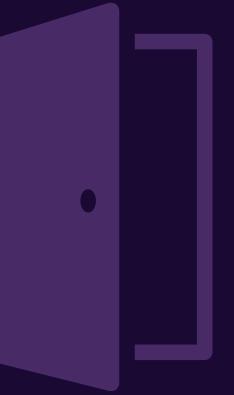


LLAMADAS AL SISTEMA PARA MANEJO DE ARCHIVOS Y DIRECTORIOS

Son mecanismos mediante el cual un programa o aplicación solicita servicios del kernel del sistema operativo para que puedan interactuar con el sistema de archivos, que es responsable de administrar archivos y directorios en los dispositivos de almacenamiento.



OPEN()



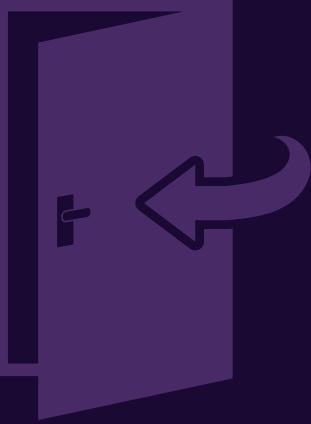
La llamada al sistema open es el primer paso que debe ejecutar todo proceso que quiera acceder a los datos de un archivo. Su sintaxis es:

```
fd = open(pathname, flags, modes);
```

donde:

- **pathname**: ruta y nombre del archivo.
- **flags**: modo de apertura, para lectura, escritura, etc.
- **modes**: permisos del archivo en caso de que se deba crear.
- **fd**: entero que representa el descriptor del archivo.

CLOSE()



Se utiliza para cerrar un archivo abierto, liberando los recursos asociados. Una vez que un archivo ya no es necesario, se debe cerrar para liberar recursos del sistema.

Su sintaxis es:

```
close(fd);
```

donde:

- **fd**: descriptor de archivo que devuelve la llamada open



READ()



Permite que un programa lea datos de un archivo abierto. Su sintaxis es:

```
number = read (fd, buffer, count);
```

donde:

- **fd**: descriptor de archivo que devuelve la llamada `open`
- **buffer**: dirección del buffer donde se van a colocar los datos leídos
- **count**: número de bytes que el usuario quiere leer
- **number**: número de bytes leídos si es positivo en caso contrario la ejecución no ha sido correcta

WRITE()



Permite que un programa escriba datos en un archivo abierto. La sintaxis es:

```
number = write (fd, buffer, count);
```

donde:

- **fd**: descriptor de archivo que devuelve la llamada `open`
- **buffer**: dirección del buffer donde se encuentran los datos a escribir
- **count**: número de bytes que el usuario quiere escribir
- **number**: número de bytes escritos si es positivo en caso contrario la ejecución no ha sido correcta



**¡GRACIAS POR
LA ATENCIÓN!**

¿Dudas?