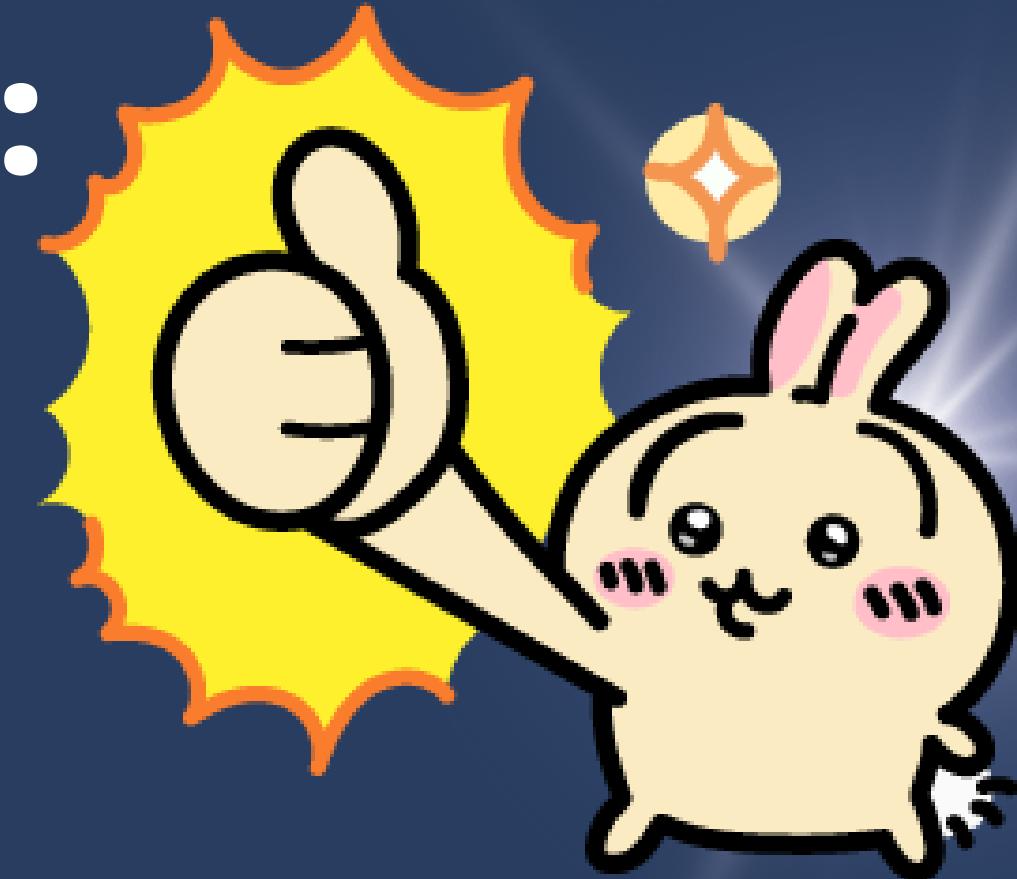


SISTEMAS OPERATIVOS

# AYUDANTÍA 1: SYSCALLS

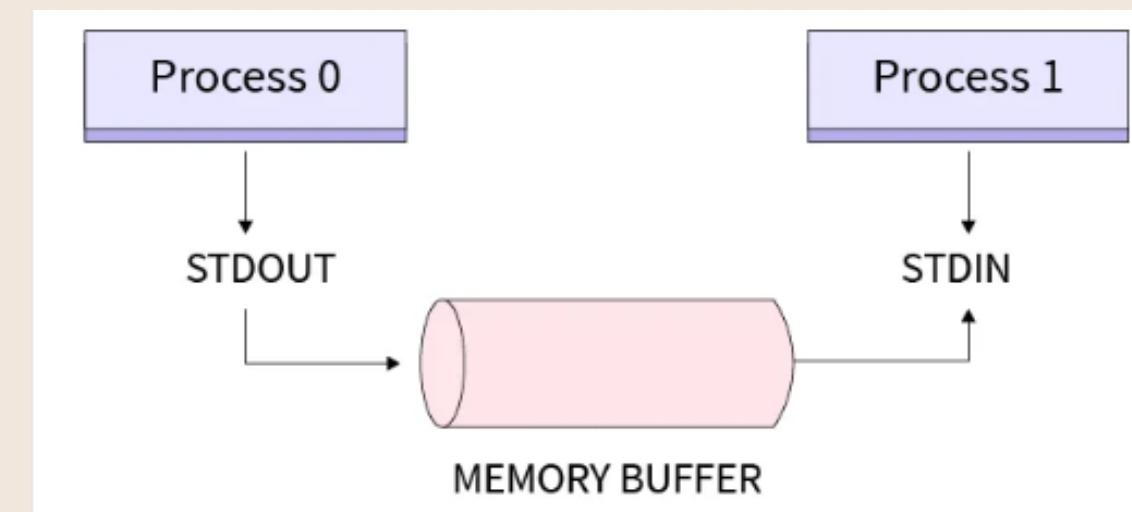


Profesor : Jonathan Frez  
Ayudante :Dante Hortuvia



## REGLAS DE LAS TAREAS

- Windows queda estrictamente prohibido en cualquier tipo de clase, ejercicio, tarea y cualquier cosa relacionada al curso
- El uso de IA generativa esta permitido, siempre y cuando no se utilice de manera abusiva
- Para la tarea 1 se deben utilizar **Named Pipes**



## • P R O C E S O S

Un proceso, es un programa que se está ejecutando y que utiliza un espacio de memoria.

### • ¿ Q U É E S E L P C B ?



El PCB (Process Control Block) : Guarda toda la información de un proceso, donde se guardan los siguientes datos:

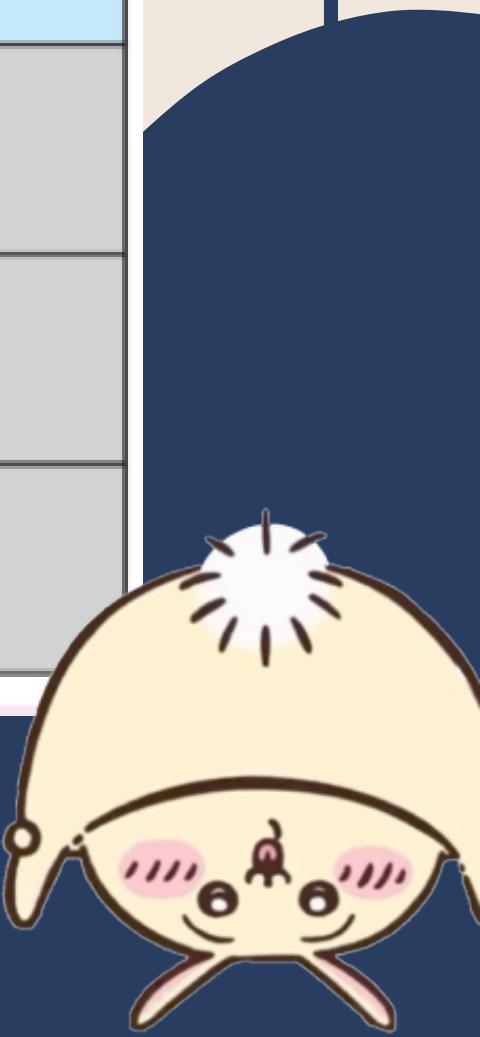
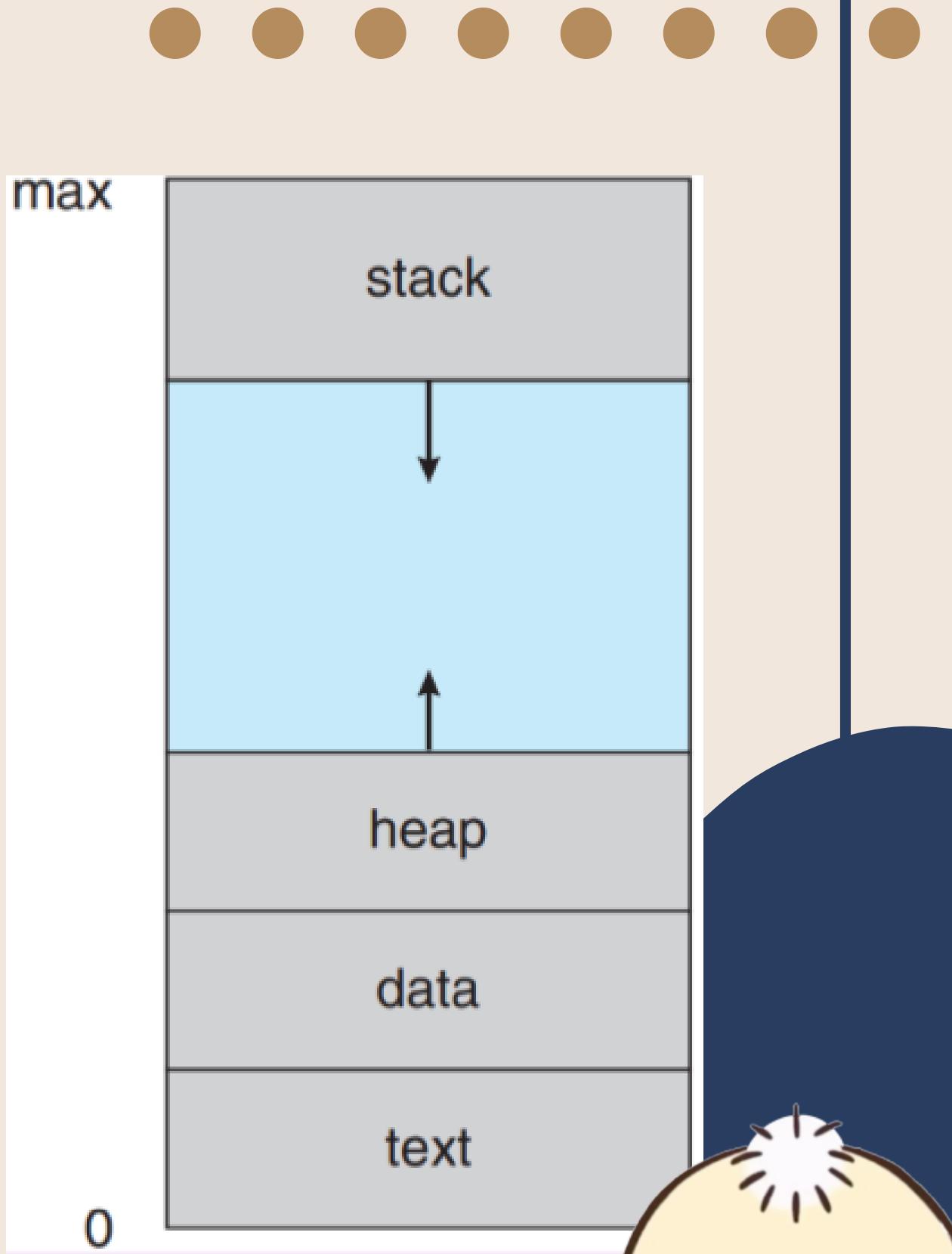
- **Estado del proceso** : El estado actual del proceso el cual puede ser Running, Ready, Waiting y Terminated.
- **Número del proceso** : El id del proceso en cuestión se denomina PID, este es único para cada proceso el cual se libera una vez terminado.
- **Program Counter** : Dirección de memoria de la próxima instrucción que debe ejecutar el CPU del proceso.
- **Límite de memoria** : Límite de la memoria asignada al proceso

PCB

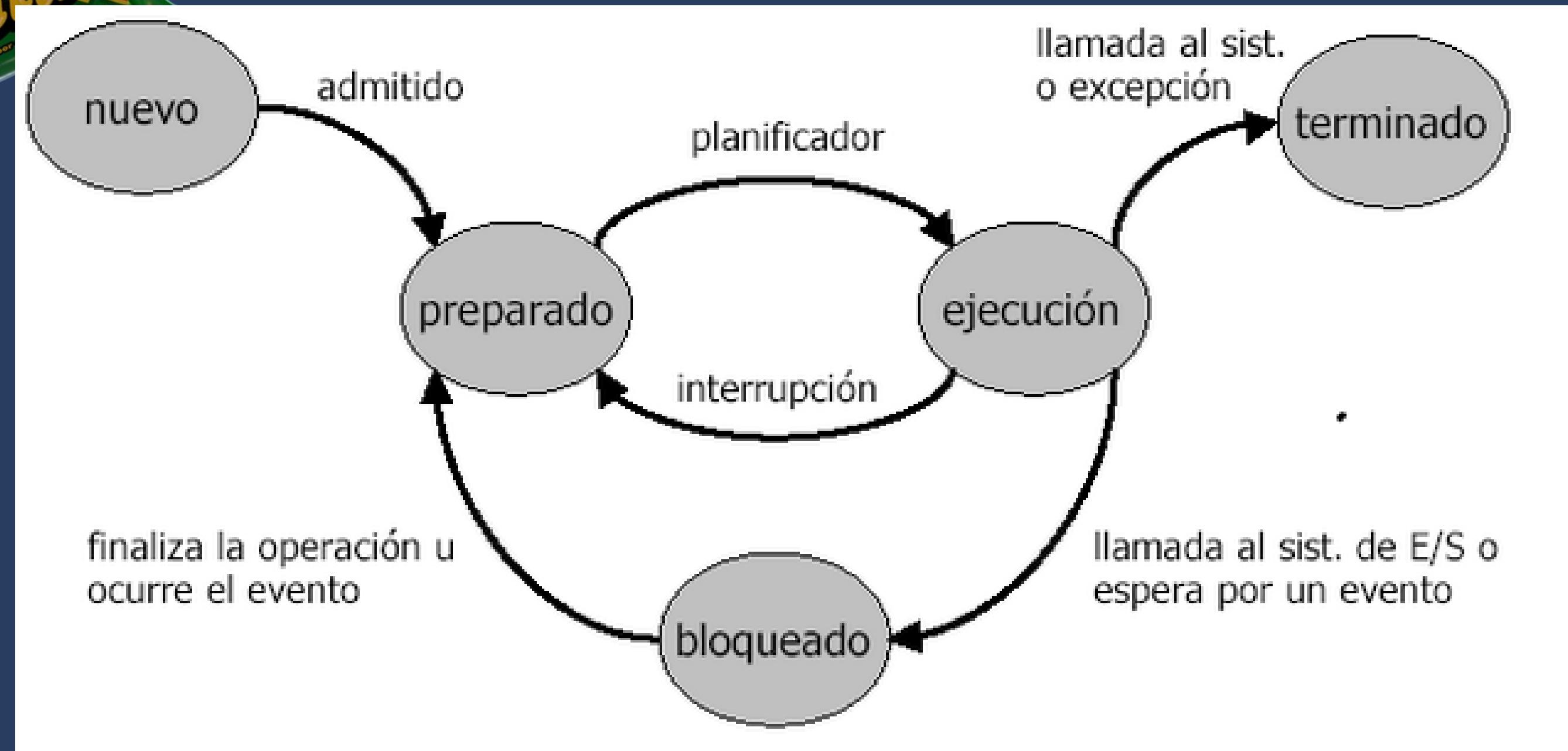
process state
process number
program counter
registers
memory limits
list of open files
• • •

# • PROCESO EN MEMORIA

- **Stack:** Guarda las variables locales entre otros registros, crece hasta abajo
- **Gap:** Espacio entre el Stack y Heap hecho para crecer según lo necesario
- **Heap:** Se usa como memoria dinámica en tiempo de ejecución (para new en C++ por ejemplo, o malloc en C), crece hacia arriba.
- **Data:** Variables globales y/o estáticas ya inicializadas.
- **Text:** El código ejecutándose en el proceso.



# Procesos



# • ESTADO DE UN PROCESO:

## • NUEVO

El proceso está en fase de creación. Se le asigna espacio en memoria y se inicializan los recursos necesarios para su ejecución.

## • LISTO:

El proceso ya cuenta con todos los recursos que necesita y está preparado para ejecutarse. Permanece en la cola de listos esperando su turno para acceder a la CPU.

## • EN EJECUCIÓN:

El proceso se encuentra actualmente ejecutándose en la CPU. En un sistema mono núcleo solo puede haber un proceso en este estado por procesador.



## • ESTADO DE UN PROCESO:

### • EN ESPERA

El proceso no puede continuar porque espera un evento externo o un recurso (por ejemplo, la fiscalización de una operación de entrada/salida). Una vez satisfecha la condición que lo bloquea, vuelve al estado Listo.

### • TERMINADO

El proceso ha concluido su ejecución. Sus recursos son liberados, aunque su entrada en la tabla de procesos puede permanecer hasta que el proceso padre recoja su estado de salida.



## • MODO DUAL

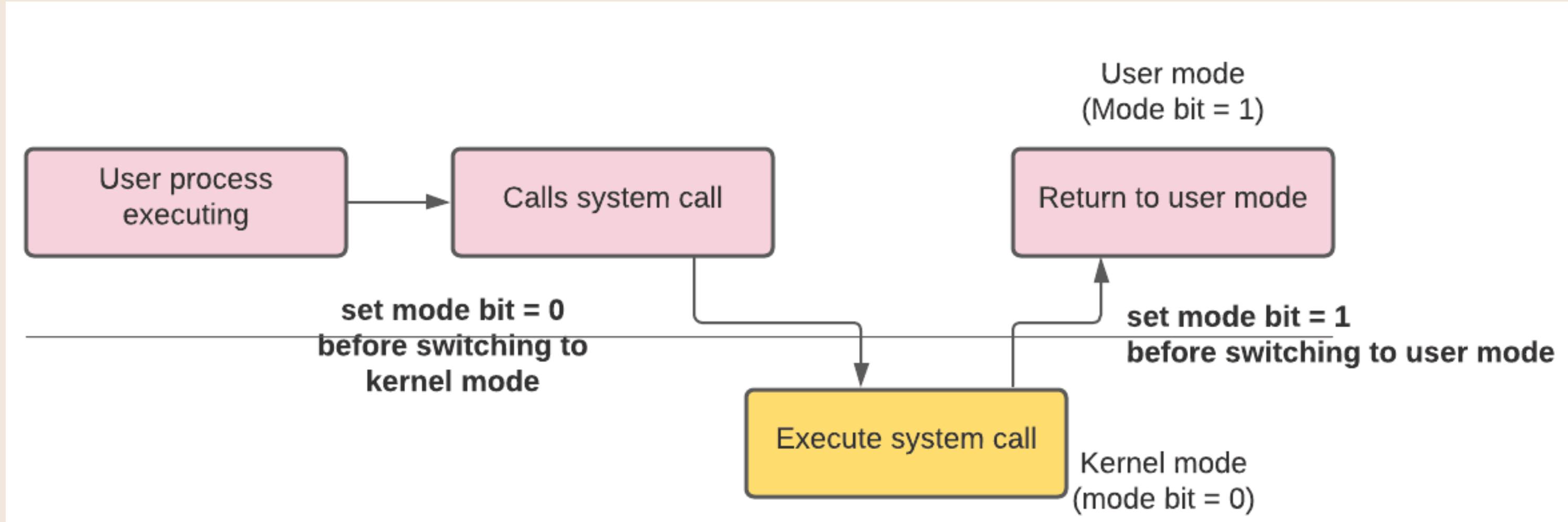
El modo dual es un mecanismo de seguridad utilizado por los sistemas operativos para controlar el acceso a los recursos del hardware. El procesador puede trabajar en modo usuario o en modo kernel.

En el primero se ejecutan las aplicaciones de los usuarios con permisos limitados, evitando que un programa pueda dañar el sistema. Cuando una aplicación necesita usar recursos críticos (como memoria, disco o dispositivos de entrada/salida), debe solicitarlo al sistema operativo mediante llamadas al sistema.

En cambio, el modo kernel es exclusivo del sistema operativo y permite ejecutar instrucciones privilegiadas con acceso total al hardware. Gracias a esta separación, se garantiza que el sistema operativo tenga el control absoluto de los recursos y se proteja la estabilidad del equipo, impidiendo que un error en un programa afecte al resto de procesos o al funcionamiento global del sistema.



## • MODO DUAL



**TRANSICION ENTRE EL MODO  
USARIO A MODE KERNEL**



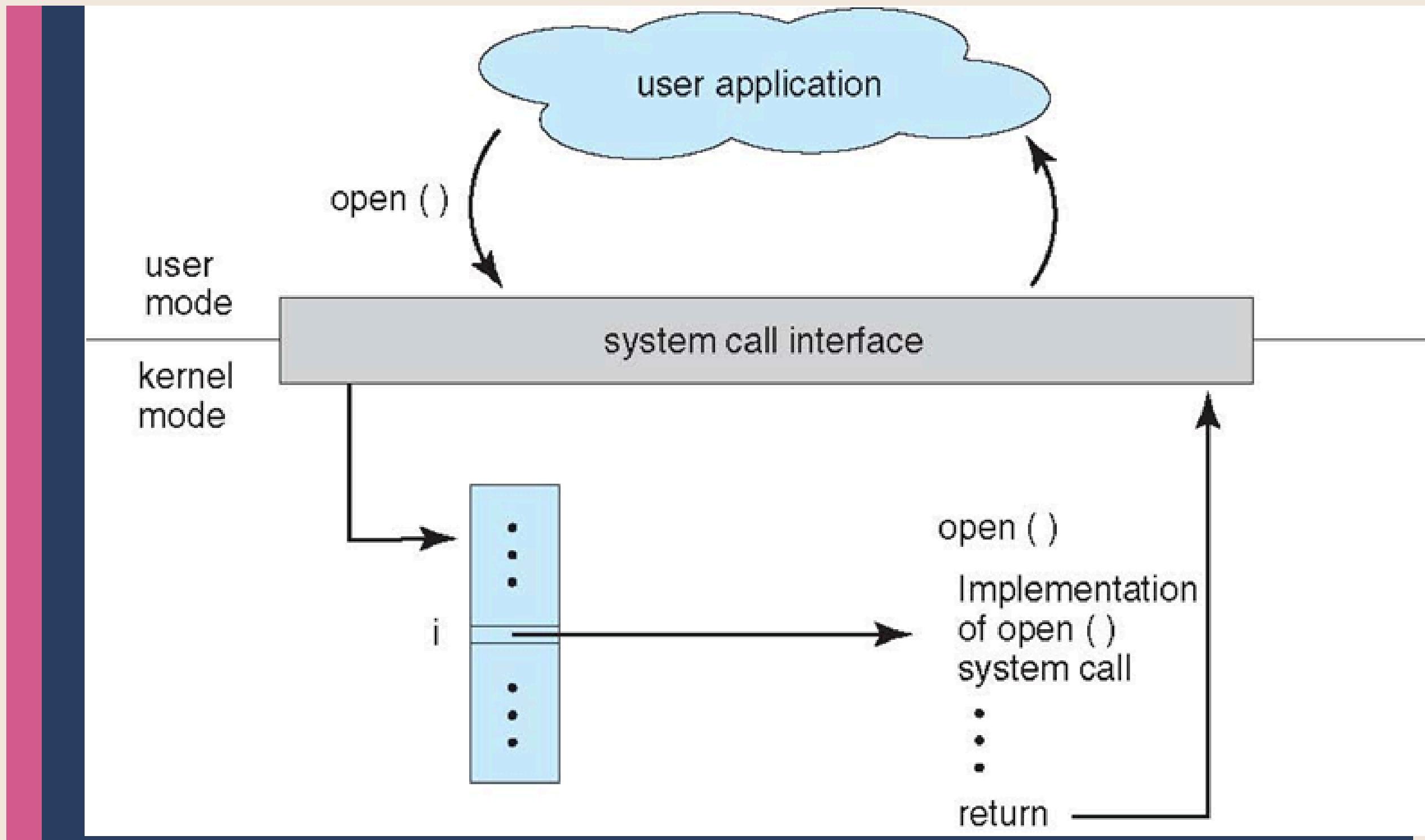
# • SYSCALLS

Las llamadas al sistema (System Calls o Syscalls) son el mecanismo mediante el cual los programas de usuario pueden solicitar servicios al sistema operativo. Funcionan como una interfaz controlada que conecta el software de aplicación con el kernel, permitiendo acceder a recursos como archivos, memoria, dispositivos de entrada/salida o la gestión de procesos.

Cada sistema operativo define su propio conjunto de syscalls, pero todos cumplen la misma función: garantizar que el acceso a los recursos del hardware sea seguro, regulado y no directo. En lugar de que los programas manipulen el hardware por sí mismos, envían peticiones al kernel a través de la System Call Interface, asegurando así la estabilidad y protección del sistema.



# • SYSCALLS



**APLICACIÓN DE  
USUARIO  
INVOCANDO LA  
SYSCALL OPEN()**



# ALGUNAS SYSCALLS

Fork()

Exec()

Wait()

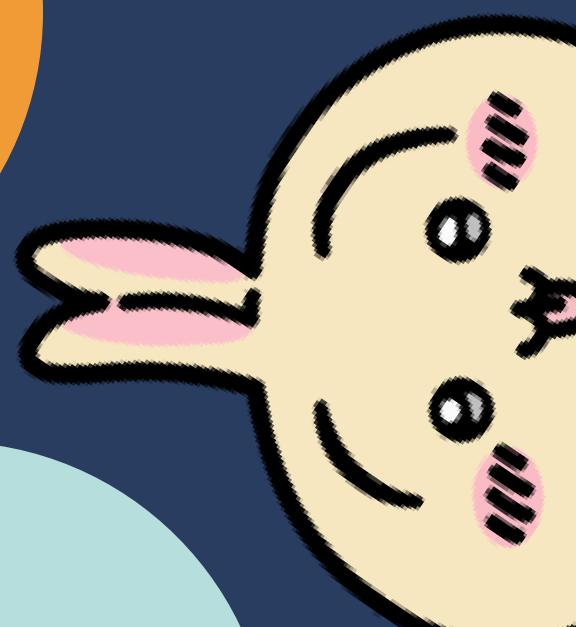
Killall()

Exit()

Kill()

Sleep()

Getpid()



# ALGUNAS SYSCALLS

`fork()`: Crea un proceso hijo duplicando al proceso padre. Ambos continúan ejecutando el mismo código, pero con PIDs distintos.

`exec()`: Sustituye el código del proceso actual por el de un nuevo programa, manteniendo el mismo PID pero cambiando completamente su contenido.

`wait()`: Hace que el proceso padre se detenga hasta que finalice uno de sus procesos hijos. Además, permite recoger el estado de salida del hijo y así evitar la creación de procesos zombis.

`exit()`: Finaliza la ejecución del proceso actual y libera todos los recursos que tenía asignados.

`kill()`: Envía una señal a un proceso específico (identificado por su PID), ya sea para terminarlo o para indicarle que ejecute alguna acción definida por esa señal.

`killall()`: Finaliza todos los procesos que coincidan con un nombre de programa determinado.

`sleep()`: Suspende la ejecución del proceso actual durante un tiempo definido (en segundos).

`getpid()`: Retorna el Process ID del proceso actual.



# Proceso Huérfano

Un proceso huérfano aparece cuando su proceso padre termina antes que él. En estos casos, el proceso huérfano es adoptado automáticamente por el proceso init (o su equivalente en el sistema operativo). Esto asegura que el proceso pueda finalizar correctamente sin generar problemas, ya que el sistema operativo se encarga de gestionarlo.

# Proceso Zombi

Un proceso zombi ocurre cuando un proceso hijo ha terminado su ejecución, pero su proceso padre aún no ha recogido su estado de salida mediante `wait()`. Aunque el proceso ya no se ejecuta, sigue ocupando un espacio en la tabla de procesos, lo que puede causar problemas si se acumulan muchos zombies y se agotan los recursos del sistema.



# Fork()



Es una syscall que crea un proceso exacto de el (Hijo), es una copia exacta del proceso padre, con la diferencia de que tienen distinto pid y que el hijo tiene un espacio de memoria propia y de ejecución.

## Tipos de respuestas:

- Si el fork() retorna -1 significa que hubo un error al momento de hacer fork().
- Si retorna un número > 0 significa que es el proceso padre
- Si retorna un número = 0 significa que es el proceso hijo.

```
int main(){
    pid_t pid = fork();
    if(pid == 0){
        printf("Hola soy el hijo y mi id es %d y mi pid es %d\n", getpid(), pid);
    }else{
        printf("Hola soy el padre y mi id es %d y mi pid es %d\n", getpid(), pid);
    }
}
```

# Variables entre procesos



```
int main(){
    int contador = 0;
    pid_t pid = fork();

    if(pid > 0){
        for(int i = 0;i < 10 ; i++){
            contador++;
            printf("Soy el proceso padre y llevo la cuenta de %d\n",contador);
        }
    }else if(pid == 0){
        for(int i = 0;i < 10 ; i++){
            contador++;
            printf("Soy el proceso Hijo y llevo la cuenta de %d\n", contador);
        }
    }
    return 0;
}
```

Cada proceso tiene un contador propio, que no afecta al otro.

¿Los procesos padre e hijo comparten el contador? ¿O cada uno tiene uno propio?

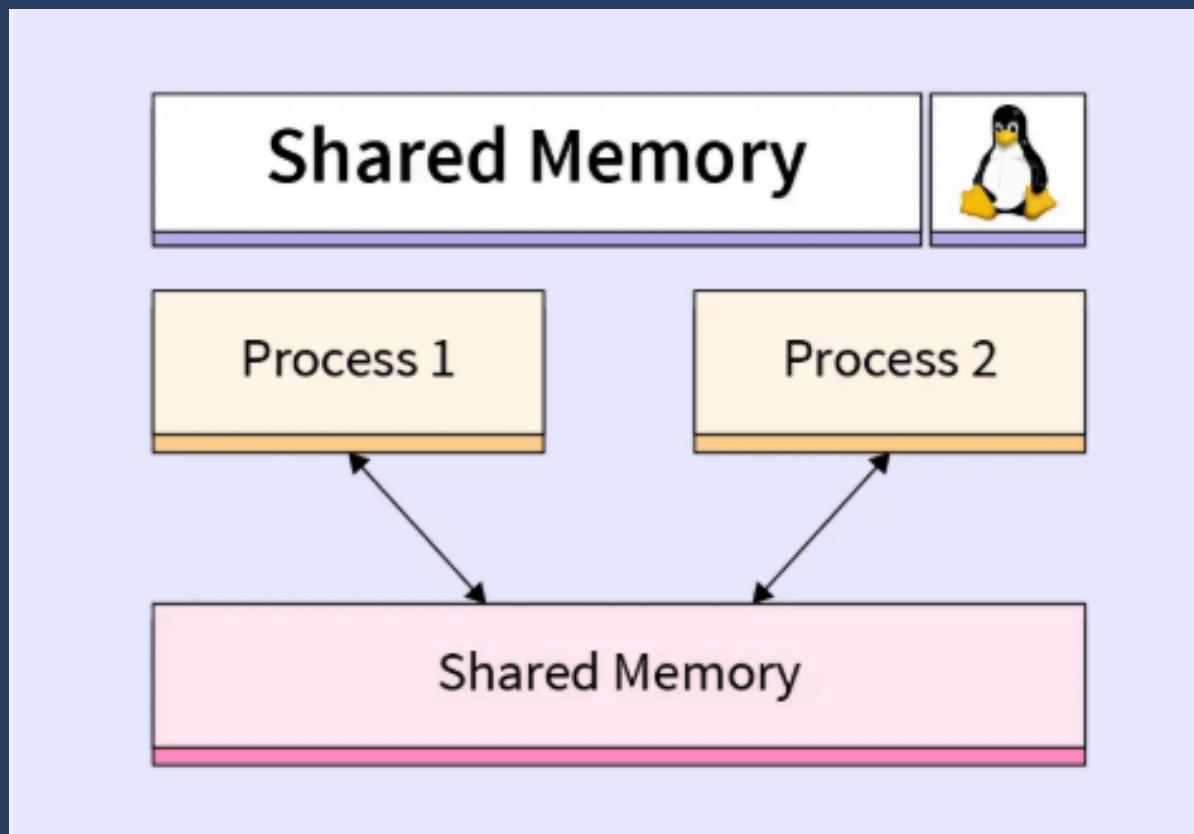
```
Soy el proceso padre y llevo la cuenta de 1
Soy el proceso Hijo y llevo la cuenta de 1
Soy el proceso padre y llevo la cuenta de 2
Soy el proceso Hijo y llevo la cuenta de 2
Soy el proceso Hijo y llevo la cuenta de 3
Soy el proceso padre y llevo la cuenta de 3
Soy el proceso Hijo y llevo la cuenta de 4
Soy el proceso padre y llevo la cuenta de 4
Soy el proceso Hijo y llevo la cuenta de 5
Soy el proceso padre y llevo la cuenta de 5
```

# ¿Cómo podemos compartir variables?



Mediante Memoria Compartida:

Una forma de comunicación entre los procesos , donde se permite compartir memoria entre ellos.



- `shmget`: se utiliza para crear o acceder a un espacio de memoria compartida, necesita una llave, un tamaño y opciones.
- `shmid` es el id de la memoria.
- `shmaddr`: se utiliza para especificar la dirección de la memoria, donde se conecta el proceso.
- `shmctl` se utiliza para realizar operaciones de control en la memoria.
- `shmdt` desconecta la memoria de un proceso.

# Ejemplo Memoria compartida

```
key_t key = 1234;//Se establece llave del espacio de memoria  
int shmid;//espacio del memoria  
  
shmid = shmget(key,sizeof(int), IPC_CREAT | 0666);  
if(shmid == -1) {//Manejo de errores  
    perror("shmget");  
    exit(EXIT_FAILURE);  
}  
int *contador = (int*) shmat(shmid, NULL, 0);//Creacion de la variable  
if(contador == (int*) -1) { // manejo de errores  
    perror("shmat");  
    exit(1);  
}  
*contador = 0;//Inicio en 0
```

```
for(int i = 0;i < 10 ; i++){  
    (*contador)++;  
    printf("Soy el proceso padre y llevo la cuenta de %d\n",*contador);  
}  
else if(pid == 0){  
    for(int i = 0;i < 10 ; i++){  
        (*contador)++;  
        printf("Soy el proceso Hijo y llevo la cuenta de %d\n",*contador);  
    }  
    if(shmdt(contador) == -1){  
        perror("shmdt");  
        exit(EXIT_FAILURE);  
    }  
    if(shmctl(shmid, IPC_RMID,NULL) == -1){  
        perror("shmctl");  
        exit(EXIT_FAILURE);  
    }
```

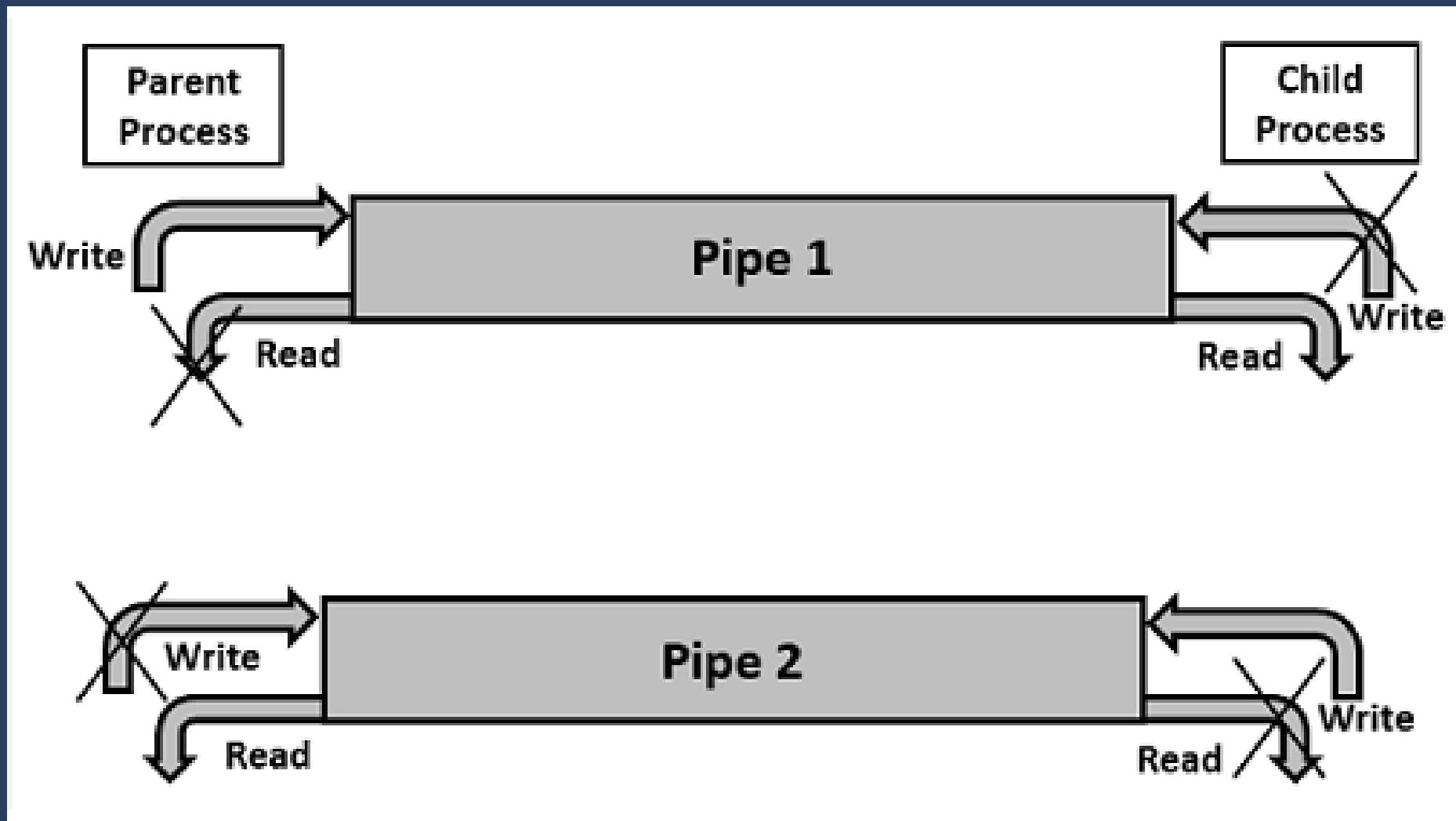


## Resultado

Soy el proceso padre y llevo la cuenta de 1  
Soy el proceso Hijo y llevo la cuenta de 2  
Soy el proceso padre y llevo la cuenta de 3  
Soy el proceso Hijo y llevo la cuenta de 4  
Soy el proceso padre y llevo la cuenta de 5  
Soy el proceso Hijo y llevo la cuenta de 6  
Soy el proceso padre y llevo la cuenta de 7  
Soy el proceso Hijo y llevo la cuenta de 8  
Soy el proceso padre y llevo la cuenta de 9  
Soy el proceso Hijo y llevo la cuenta de 10  
Soy el proceso padre y llevo la cuenta de 11  
Soy el proceso Hijo y llevo la cuenta de 12

# Pipes

Pipes anónimos: Tienen una comunicación unidireccional entre dos procesos, se utiliza entre los procesos padre e hijo.



- Para crear pipes se hace atreves de la syscall pipe(), se pasa un array de dos elementos tipo int.
- se utiliza 0 para leer (READ) y 1 para escribir (WRITE).
- Las dos syscall necesitan : Archivo que represente al pipe. Espacio de memoria. Tamaño.



# Ejemplo Pipes sin nombre

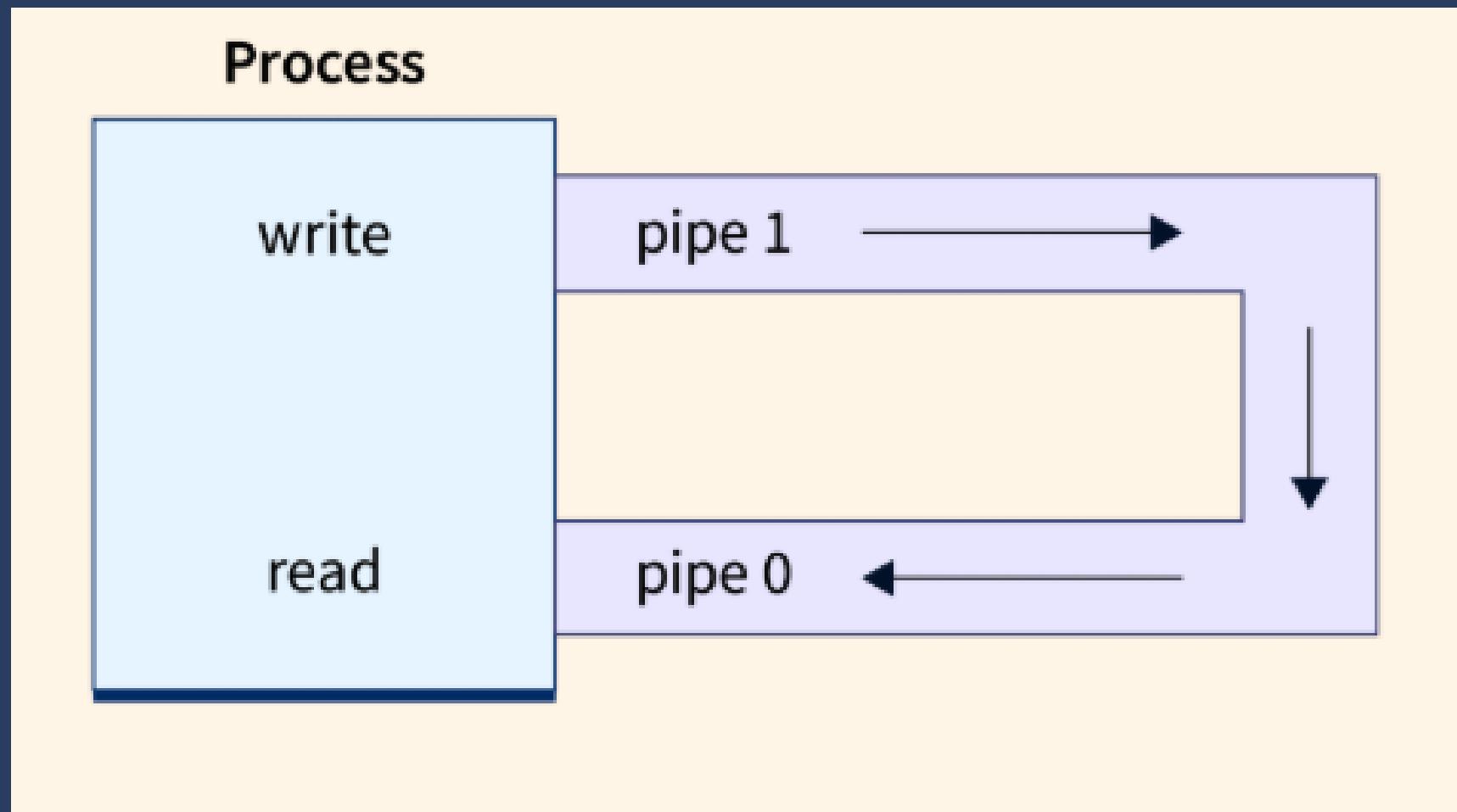


```
int main(){
    int fd1[2];
    pid_t pid;
    pipe(fd1); // Creacion de pipes
    pid = fork();

    if(pid == 0){
        close(fd1[1]);
        char buffer[100];
        int Buffer_read = read(fd1[0], buffer, sizeof(buffer));
        if(Buffer_read == -1){ // Manejo de errores
            perror("READ PIPE");
            exit(-1);
        }
        buffer[Buffer_read] = '\0'; // señal de termino
        printf("Llegaron las %s\n",buffer);
    } else {
        close(fd1[0]);
        char *mensaje = "Pipshass";
        ssize_t bytes_escrito = write(fd1[1],mensaje ,strlen(mensaje));
        if(bytes_escrito == -1){
            perror("WRITE PIPE");
            exit(-1);
        }
        printf("Enviado\n");
        close(fd1[1]);
    }
}
```

# Pipes con nombre

Pipes FIFO (Con nombre) : Tienen una comunicación bidireccional, no requieren una ser padre e hijo.



- Se crean con la syscall mkfifo(), se le tiene que dar un nombre y respectivos permisos en octal.
- Tiene read() y write()
- Eliminar al final de la ejecucion el pipe, ya que este permanece luego de haber finalizado la ejecución.



**Deben utilizarlo  
en la tarea**

# Ejemplo Pipes con Nombre

```
int main(){
    const char *fifo_path = "./my_fifo2"; // Nombre de la ruta del pipe

    int fd;
    int num;

    mkfifo(fifo_path, 0666); // Creacion del pipe
    fd = open(fifo_path, O_WRONLY); // Abertura del pipe

    printf("Ingrese un numero para multiplicar por 100\n");
    scanf("%d", &num);

    write(fd, &num, sizeof(num)); // Envio de mensaje
    close(fd);
    unlink(fifo_path);

    return 0 ;
}
```

Código 1



```
int main(){
    const char *fifo_path = "./my_fifo2"; // Nombre de la ruta

    int fd;
    int num;

    mkfifo(fifo_path, 0666); // Creacion del pipe
    fd = open(fifo_path, O_WRONLY); // Abertura del pipe

    read(fd, &num, sizeof(num)); // recibo del mensaje

    printf("su numero multiplicado por 100 es: %d\n", num);

    close(fd);
    unlink(fifo_path);

    return 0 ;
}
```

Código 2

# EJERCICIOS



# Ejercicio 1

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>

int counter = 0; //variable global
int main(){ //codigo1
    pid_t r;
    while(counter < 2){
        r= fork();
        if(r > 0 ){
            execl("./Ejemplo2", "", NULL);
        }
        counter = counter + 1;

        printf("%d\n", counter);
    }
    return 0;
}
```

```
juanito_1 / Códigos_Ejercicios / Ejemplo2.c
1
2 #include <stdio.h>
3 #include <unistd.h>
4 #include <sys/types.h>
5 #include <stdlib.h>
6 #include <string.h>
7
8
9 int counter = 0; //variable global
0 int main(){ //codigo2
1     pid_t t;
2     t = fork();
3     if( t > 0 ){
4         counter = counter * 2;
5     }
6     else {
7         counter = counter -1;
8     }
9     printf("%d\n", counter);
0
1 }
2 }
```

- ¿Cuántos procesos se crean en total? Justifique.
- Explique la razón detrás de la existencia de múltiples salidas a partir de la ejecución.  
Ejemplifique con dos salidas posibles

# Ejercicio 2

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[]){
    if(fork() == fork()){
        printf("Thanos\n");
    }else{
        printf("Avengers\n")
    }
    return 0;
}
```

- ¿Cuántos Procesos se crean?
- ¿Cual puede ser una salida del código?

# Ejercicio 3

```
int main(){
    pid_t id = fork();
    for(int i = 0; i < 2 ; i++)
    {
        pid_t pid = fork();
        if(pid == id){
            printf("Hola como estas\n");
        }else{
            printf("Adios\n");
        }
    }
}
```

- Cuántos procesos se crean?
- ¿Cuál puede ser una respuesta del código?



# Ejercicio 4

```
int main() {
    // Código 1
    pid_t r[3];

    for (int i = 0; i < 3; i++) {
        r[i] = fork();
        if (r[i] == 0) {
            printf("Ramen\n");
            execlp("./Codigo2", "", NULL);
        }
    }

    sleep(10);
    printf("Sushi\n");

    for (int i = 0; i < 3; i++) {
        kill(r[i], SIGKILL);
    }

    return 0;
}
```

```
int main() { // Código 2
    sleep(666);
    fork();
    printf("Brisket\n");
    exit(0);
}
```

- Suponga que existe una justa asignación de tiempo de CPU entre los procesos del sistema. ¿Cuántos procesos se crean en total? ¿Cuáles son las posibles salidas? Justifique su respuesta.

# Ejercicios 5

```
int num = 1; // variable global

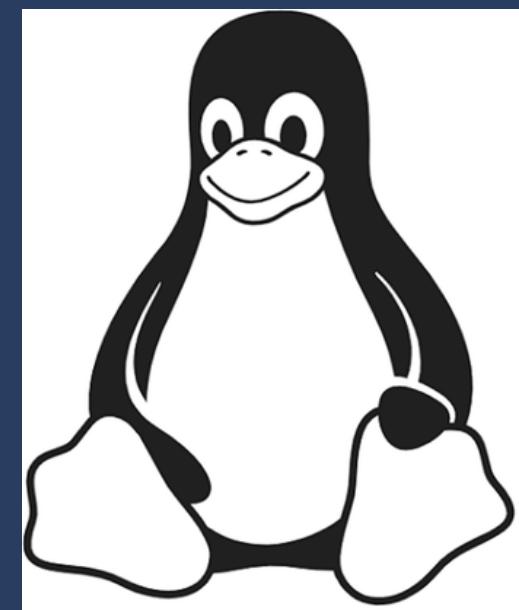
int main() {
    pid_t t = fork();
    if (t != 0) {
        num = num * 2;
        pid_t tt = fork();
        if (tt > 0) {
            num = num - 2;
        } else if (tt < 0) {
            fork();
            num = num + 2;
        }
    }
    num = num + 1;
    sleep(1);
    printf("%d\n", num);
}
```

- ¿Cuántos procesos se crean en total?
- Indique dos posibles salidas.
- Suponga ahora que por una razón misteriosa la primitiva fork() de la línea 6 falla para todos los posibles procesos que ejecutan dicha instrucción. ¿Cuál(es) sería(n) la(s) posible(s) salida(s)?

# Ejercicio 6

```
int main() {
    if (fork() != fork()) {
        pid_t t = fork();
        printf("Artorias\n");
        if (t > 0) {
            printf("Ornstein\n");
        } else if (t == 0) {
            printf("Smough\n");
            exit(0);
        } else {
            printf("Hydra\n");
            exit(0);
        }
    }
    printf("Gwyn\n");
}
```

- ¿Cuántos procesos se crean en total?  
Justifique su respuesta con un dibujo que represente el árbol de procesos generado.
- Indique cuántas veces se imprime cada una de las posibles salidas del código.



# Ejercicio 7

```
inf main(){
    pid_t r,s,t,p;
    r = fork();
    s = fork();
    t = fork();
    p = getpid();
    printf("r: %d\n",r);
    printf("s: %d\n",s);
    printf("t: %d\n",t);

    if(p%2 == 0){
        exit(0);
    }else{
        execlp("./Y","","NULL");
    }
    return 0;
}
```

```
int main(){
    pid_t m,n;
    m = getpid();

    if(m%2 == 1) {
        exit(0);
    }else {
        n = fork();
        execvp("./X","","NULL");
    }
    return 0;
}
```

- Tome como referencia el primer process id: 4331. Escriba una tabla de posibles valores para r, s y t al ejecutar ./X. Recuerde tomar en cuenta todos los procesos que surgen del primer paso por ./X
- Mencione cuantos procesos terminan posterior a haber ejecutado código del proceso Y, y justifique detalladamente
- Explique qué sucedería si se cambia la expresión “m%2 == 1” por “m%2 == 0” en el código del proceso Y y se ejecuta ./X. Describa detalladamente y justifique porque se ejecutan los procesos en la forma en que sucede.



# Ejercicio 8

Martín Gutiérrez, profesor de su sección de S00000 este semestre, tiene hijos gemelos. Estos bebés deben prepararse, antes de ir a la guardería en la mañana. A su vez, el viejo se prepara y lleva a los niños en auto a la guardería. Considere que los niños toman biberón, se bañan y se visten y que el viejo toma desayuno, se ducha, se viste y prepara el auto para que todos se puedan ir. Luego, lleva a los niños a la guardería y se va al trabajo.

En este contexto descrito, considerando que cada acción es una función y teniendo en cuenta TODAS las syscalls referidas a procesos vistas en clases, escriba los códigos de procesos Viejo y Gemelo de acuerdo con la dinámica expuesta.



# Contacto



DANTE.HORTUVIA@MAIL.UDP.CL



+56 9 2236 9606



[HTTPS://GITHUB.COM/DOSHUERTOS/AYUDANTIAS](https://github.com/doshuertos/ayudantias)

[SO\\_2\\_2025.GIT](#)



# Gracias

