

# **SISTEMAS OPERATIVOS**

Misericordia Avila Irigaray

# TEMA - 1

User mode: Llamadas a sistemas.

Kernel mode: Modo privilegiado: Excepciones (síncronas / se resuelven en la misma instrucción si se puede), interrupciones (asíncronas), llamadas a sistemas (síncronas).

Hay partes de la memoria sólo accesibles en modo privilegiado y determinadas instrucciones de lenguaje máquina sólo se pueden ejecutar en modo privilegiado.

Las **llamadas a sistema** son un conjunto de FUNCIONES que ofrece el kernel para acceder a sus servicios.

Ejemplo:

Librería de C: printf en lugar de write

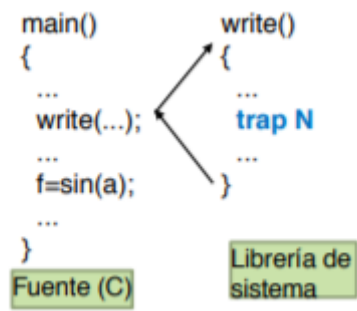
Nota: La librería se ejecuta en modo usuario y no puede acceder al dispositivo directamente

Requieren:

- Se deben salvar/restaurar los registros modificados
- Ejecución en modo privilegiado <- soporte HW
- Paso de parámetros y retorno de resultados entre modos de ejecución diferentes <- depende HW
- Las direcciones que ocupan las llamadas a sistema tienen que poder ser variables para soportar diferentes versiones de kernel y diferentes S.O <- por portabilidad

La librería de sistema se encarga de traducir de la función que ve el usuario a la petición de servicio explícito al sistema

- Pasa parámetros al kernel
- Invoca al kernel <- TRAP
- Recoge resultados del kernel
- Homogeneiza resultados (todas las llamadas a sistema en linux devuelven -1 en caso de error)



# TEMA – 2

## 1 - PROCESOS

Un proceso es la representación del SO de un programa en ejecución.

Para gestionar la información de un proceso, el sistema utiliza una estructura de datos llamada **PCB** (Process Control Block).

El PCB contiene (en el kernel):

- Espacio de direcciones: descripción de las regiones del proceso: código, datos, pila, ...

- Contexto de ejecución:

SW: PID, información para la planificación, información sobre el uso de dispositivos, estadísticas...

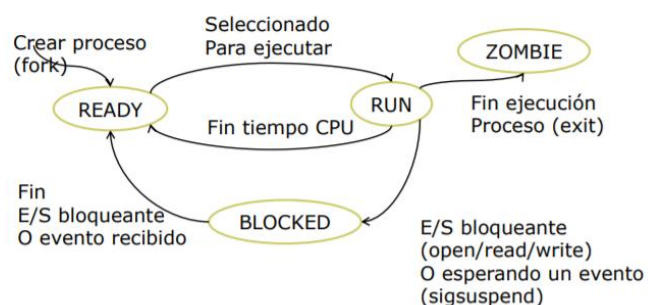
HW: tabla de páginas, program counter...

Procesos **secuenciales**: Uno detrás del otro.

Procesos **concurrentes**: Todos a la vez.

El **estado** suele gestionarse o con un campo en el PCB o teniendo diferentes listas o colas con los procesos en un estado concreto:

- **Run**: El proceso tiene asignada una CPU y está ejecutándose
- **Ready**: El proceso está preparado para ejecutarse, pero está esperando que se le asigne una CPU
- **Blocked**: El proceso no tiene/consume CPU, está bloqueado esperando un que finalice una entrada/salida de datos o la llegada de un evento
- **Zombie**: El proceso ha terminado su ejecución, pero aún no ha desaparecido de las estructuras de datos del kernel



Un proceso contiene en su PCB:

■ Identidad:

- Define quién es (identificador, propietario, grupo) y qué puede hacer el proceso (recursos a los que puede acceder)

- PID: Identificador único para el proceso

■ Entorno:

- Parámetros (argv en un programa en C) y variables de entorno (HOME, PATH, USERNAME, etc)

■ Contexto:

- Toda la información que define el estado del proceso, todos sus recursos que usa y que ha usado durante su ejecución.

## **FORK**

El hijo es un duplicado del padre (UNIX), pero cada uno tiene su propia memoria física. Además, padre e hijo, en el momento de la creación, tienen el mismo contexto de ejecución (los registros de la CPU valen lo mismo).

El hijo:

■ HEREDA:

- El espacio de direcciones lógico (código, datos, pila, etc). La memoria física es nueva, y contiene una copia de la del padre (en el tema 3 veremos optimizaciones en este punto - COW)
- La tabla de programación de signals
- Los dispositivos virtuales
- El usuario /grupo (credenciales)
- Variables de entorno

■ NO HEREDA: (sino que se inicializa con los valores correspondientes)

- PID, PPID (PID de su padre)
- Contadores internos de utilización (Accounting)
- Alarmas y signals pendientes (son propias del proceso)

## WAITPID

Si queremos sincronizar el padre con la finalización del hijo, podemos usar waitpid: El proceso espera (si es necesario se bloquea el proceso) a que termine un hijo cualquiera o uno concreto.

-waitpid(-1,NULL,0) <- Esperar (con bloqueo si es necesario) a un hijo cualquiera

-waitpid(pid\_hijo,NULL,0) <- Esperar (con bloqueo si es necesario) a un hijo con pid=pid\_hijo

## EXIT

Cuando un proceso quiere finalizar su ejecución (voluntariamente), liberar sus recursos y liberar las estructuras de kernel reservadas para él, se ejecuta la llamada a sistema exit.

## EXCELP

Mutación del proceso.

## 2 - SIGNALS

- El proceso puede **capturar** (modificar el tratamiento asociado) todos los tipos de signals excepto **SIGKILL y SIGSTOP**.

Nombre	Acción Defecto	Evento
<b>SIGCHLD</b>	IGNORAR	Un proceso hijo ha terminado o ha sido parado
<b>SIGCONT</b>		Continúa si estaba parado
<b>SIGSTOP</b>	STOP	Parar proceso
<b>SIGINT</b>	TERMINAR	Interrumpido desde el teclado (CtrlC)
<b>SIGALRM</b>	TERMINAR	El contador definido por la llamada alarm ha terminado
<b>SIGKILL</b>	TERMINAR	Terminar el proceso
<b>SIGSEGV</b>	CORE	Referencia inválida a memoria
<b>SIGUSR1</b>	TERMINAR	Definido por el usuario (proceso)
<b>SIGUSR2</b>	TERMINAR	Definido por el usuario (proceso)

Al recibir un signal, el proceso interrumpe la ejecución del código, pasa a ejecutar el tratamiento que ese tipo de signal tenga asociado y al acabar (si sobrevive) continúa donde estaba.

Los procesos pueden **bloquear/desbloquear** la recepción de cada signal excepto SIGKILL y SIGSTOP (tampoco se pueden bloquear los signals SIGFPE, SIGILL y SIGSEGV si son provocados por una excepción).

- Cuando un proceso bloquea un signal, si se le envía ese signal el proceso no lo recibe y el sistema lo marca como pendiente de tratar = **bitmap asociado al proceso, sólo recuerda un signal de cada tipo.**
- Cuando un proceso desbloquea un signal, recibirá y tratará el signal pendiente de ese tipo.

Servicio	Llamada sistema
Enviar un signal concreto	kill
Capturar/reprogramar un signal concreto	sigaction
Bloquear/desbloquear signals	sigprocmask
Esperar HASTA que llega un evento cualquiera (BLOQUEANTE)	sigsuspend
Programar el envío automático del signal SIGALRM (alarma)	alarm

- sigemptyset: inicializa una máscara sin signals

```
int sigemptyset(sigset_t *mask)
```



- sigfillset: inicializa una máscara con todos los signals

```
int sigfillset(sigset_t *mask)
```



- sigaddset: añade el signal a la máscara que se pasa como parámetro

```
int sigaddset(sigset_t *mask, int signum)
```



- sigdelset: elimina el signal de la máscara que se pasa como parámetro

```
int sigdelset(sigset_t *mask, int signum)
```



- sigismember: devuelve cierto si el signal está en la máscara

```
int sigismember(sigset_t *mask, int signum)
```



- La gestión de signals es por proceso, la información de gestión está en el PCB

- Cada proceso tiene una tabla de programación de signals (1 entrada por signal) <- Se indica que acción realizar cuando se reciba el evento
- Un bitmap de eventos pendientes (1 bit por signal) <- No es un contador, actúa como un booleano

- Un único temporizador para la alarma <- Si programamos 2 veces la alarma solo queda la última
- Una máscara de bits para indicar qué signals hay que tratar

## SIGACTION

■ struct sigaction: varios campos. Nos fijaremos sólo en 3:

- sa\_handler: puede tomar 3 valores
  - SIG\_IGN: ignorar el signal al recibirlo
  - SIG\_DFL: usar el tratamiento por defecto
  - Función de usuario con una cabecera predefinida: void nombre\_funcion(int s);
    - IMPORTANTE: la función la invoca el kernel. El parámetro se corresponde con el signal recibido (SIGUSR1, SIGUSR2, etc), así se puede asociar la misma función a varios signals y hacer un tratamiento diferenciado dentro de ella.
- sa\_mask: signals que se añaden a la máscara de signals que el proceso tiene bloqueados
  - Si la máscara está vacía sólo se añade el signal que se está capturando
  - Al salir del tratamiento se restaura la máscara que había antes de entrar
- sa\_flags: para configurar el comportamiento (si vale 0 se usa la configuración por defecto). Algunos flags:
  - SA\_RESETHAND: después de tratar el signal se restaura el tratamiento por defecto del signal.
  - SA\_RESTART: si un proceso bloqueado en una llamada a sistema recibe el signal se reinicia la llamada que lo ha bloqueado.

## SIGPROCMASK

int sigprocmask(int operacion, sigset\_t \*mascara, sigset\_t \*vieja\_mascara)

- Operación puede ser:



- SIG\_BLOCK: añadir los signals que indica mascara a la máscara de signals bloqueados del proceso
- SIG\_UNBLOCK: quitar los signals que indica mascara a la máscara de signals bloqueados del proceso
- SIG\_SETMASK: hacer que la máscara de signals bloqueados del proceso pase a ser el parámetro mascara

## SIGSUSPEND

int sigsuspend(sigset\_t \*mascara)

- Mientras el proceso está bloqueado en el sigsuspend, máscara será los signals que no se recibirán (signals bloqueados)
  - Así se puede controlar qué signal saca al proceso del bloqueo
- Al salir de sigsuspend, automáticamente se restaura la máscara que había y se tratarán los signals pendientes que se estén desbloqueando

## FORK Y EXECLP CON SIGNALS

### ■ FORK: Proceso nuevo

- El hijo hereda la tabla de acciones asociadas a los signals del proceso padre
- La máscara de signals bloqueados se hereda
- Los eventos son enviados a procesos concretos (PID's), el hijo es un proceso nuevo <- La lista de eventos pendientes se borra (tampoco se heredan los temporizadores pendientes)

### ■ EXECLP: Mismo proceso, cambio de ejecutable

- La tabla de acciones asociadas a signals se pone por defecto ya que el código es diferente
- Los eventos son enviados a procesos concretos (PID's), el proceso no cambia <- La lista de eventos pendientes se conserva
- La máscara de signals bloqueados se conserva

## ROUND ROBIN

### ■ Eventos que activan la política Round Robin:

1. Cuando el proceso se bloquea (no preemptivo)
2. Cuando termina el proceso (no preemptivo)
3. Cuando termina el quantum (preemptivo)

### ■ Es una política apropiativa o preemptiva

■ Cuando se produce uno de estos eventos, el proceso que está run deja la la cpu y se selecciona el siguiente de la cola de ready.

- Si el evento es 1, el proceso se añade a la cola de bloqueados hasta que termina el acceso al dispositivo
- Si el evento es el 2, el proceso pasaría a zombie en el caso de linux o simplemente terminaría
- Si el evento es el 3, el proceso se añade al final de la cola de ready

## EJEMPLOS

Llamada a sistema	Secuencia de Inodos y bloques accedidos (memoria o disco)	Accesos que requieren ir a disco
3.open	Inodo /(0)+ bloque /(0)+ inodo usr(1)+ bloque usr(1)+ inodo B(4)+ bloque B(5)+ Inodo / (0)+ bloque / (0)+ inodo home (2)+ bloque home (2)+ inodo A (5)	Inodo /(0)+ bloque /(0)+ inodo usr(1)+ bloque usr(1)+ inodo B(4)+ bloque B(5)+ inodo home (2)+ bloque home(2) + inodo A(5)
4.open	Inodo / (0)+ bloque /(0)+ inodo home (2)+ bloque home (2)+inodo B (3)	Inodo B(3)
5.lseek	Inodo B (3)	Ninguno, está en la tabla de inodos
9.close	Inodo A (5)	Inodo A(5) para actualizarlo en disco
10.close	Inodo B (3)	Inodo B(3) para actualizarlo en disco

- 4) Indica cómo se identifica un dispositivo de entrada/salida en cada uno de los niveles vistos en clase para Linux.

Nivel FISICO: mediante dos números mayor (que indicaría el tipo de dispositivo) y menor (indica dispositivo concreto respecto al mayor)

Nivel LOGICO: mediante un nombre de fichero

Nivel VIRTUAL: mediante un identificador de canal, conseguido mediante llamadas a sistema de vinculación a dispositivos (open, pipe, socket,...)

- 1) Un proceso está bloqueado en la siguiente llamada a sistema, escribiendo en una pipe. Los parámetros son correctos y bien declarados.

```
ret = write(fd_pipe[1], buffer, 500);
```

Al cabo de un rato, write se desbloquea devolviendo error, ¿a qué puede ser debido?

El último proceso lector de esa pipe la ha cerrado. Si el proceso tiene capturado el signal SIGPIPE ( y la RAS no hace acabar al proceso), la llamada write devuelve error si un proceso intenta escribir en una pipe sin lectores (errno=EPIPE)

2. (0,5 puntos) ¿En qué caso una llamada write(..) puede provocar la recepción de un signal?

Cuando intenta escribir sobre una pipe sin lectores y el signal SIGPIPE está capturado y rutina de atención no acaba el proceso.

3. (0,5 puntos) Supón que el fichero A.txt existe y que tienes permiso de lectura y escritura tanto sobre A.txt como sobre el directorio en el que está. Indica un caso en que el comando "rm -f A.txt" no elimina del sistema de ficheros el contenido del fichero.

En caso de que exista otro nombre (hardlink) para el fichero A.txt

quedan y cual es el motivo.

Id. proceso	¿Acaba? (Si/No)	Línea bloqueo	Causa
P	No	20	Su hijo nunca acaba la ejecución
H1	No	17	El read nunca le devuelve 0 porque hay posibles escritores en la pipe de la que lee (P)
H1.1	No	24	El read nunca le devuelve 0 porque hay posibles escritores en la pipe de la que lee (H1)

b) ¿Qué determinará la finalización de la ejecución recursiva del programa?

La memoria se puede agotar o llegará un momento donde el sistema se quedaría sin PIDs disponibles.

- 
1. (0,5 puntos) En un sistema de 32 bits que no dispone de carga bajo demanda y con 1 GB de memoria física, ¿podemos cargar un proceso con un espacio lógico de 2 GB?

Sí, en caso de que el sistema disponga de espacio de SWAP suficiente para albergar el proceso completo (repartido entre memoria y SWAP).

En caso contrario, no se podría cargar el proceso.

11. ret=fork());

12. resultado=sbrk(cantidad\_datos\*sizeof(int));

DATA DIFERENTE PADRE E HIJO

SI ESTUVIERA AL REVÉS, EL TAMAÑO DE LA DATA SERIA COMPARTIDO

```
write(fd, &num, sizeof(int)); //num es una variable integer
```

Superbloque (en memoria) para pedir nuevo BD, BD(10) (a disco) para escribir nuevo integer y I(6) (en Tabla de I-nodos) para actualizar información del I-nodo

situaciones:

```
res = waitpid (-1, &status,WNOHANG);
```

a.- Si el proceso no tiene hijos

waitpid retorna -1, error.

b.- Si el proceso tiene 1 hijo en estado BLOCKED y otro en estado READY

waitpid retorna 0, sin bloquearse

c.- Si el proceso tiene 3 hijos en estado ZOMBIE

retornará (sin bloquearse) en res el pid de uno de esos hijos (probablemente el primero en acabar y pasar a ZOMBIE) y en status el código de finalización de ese hijo.

4. (0,75 puntos) Supongamos ahora que añadimos la línea 6. Indica la cantidad de memoria física necesaria para tener todos los procesos en memoria, suponiendo que ya se ha ejecutado la llamada a execlp, distinguiendo por regiones y proceso

```
1. /*prog1.c */
2. char *p;
3. main(int argc, char *argv[]) {
4.     p=sbrk(1000);
5.     fork();
6.     execlp("./prog2", "prog2", (char *)0);
7.     /* código original */
8. }
```

Región y proceso	Memoria física necesaria
Datos padre	4KB
Pila padre	4KB
Código padre	4KB
Heap padre	0KB
Datos hijo	4KB
Pila hijo	4KB
Código hijo	4KB
Heap hijo	0KB

Justificación: después de mutar el espacio de direcciones que tendrán cargado tanto padre como hijo es el de prog2, y el de prog1 desaparece liberándose así la memoria que ocupaba.

el directorio /home/so1: "ln mis\_numeros.txt otros\_numeros.txt". Este comando crea un hardlink. Indica si las siguientes afirmaciones son ciertas/falsas y justifica brevemente:

i. ¿Reserva un nuevo inodo?

No, ya que estamos creando un hard link, que es solo un nuevo nombre de fichero

6

## Segundo control de teoría: QT2017-2018

Nombre alumno:

DNI:

ii. ¿Modifica el superbloque?

No, no reserva bloque de datos nuevo ni inodo nuevo

iii. ¿Modifica el bloque de datos del directorio /home/so1?

Si, ya que hay que añadir un nuevo nombre de fichero

iv. ¿Modifica el inodo de /home/so1?

Si, ya que aumenta el tamaño del directorio

- 
2. Tenemos un SO que no dispone de estado BLOCKED en el ciclo de vida de los procesos. Explica si será necesario cambiar la implementación de las llamadas:

- a) alarm: No hace falta hacer ningún cambio, ya que esta llamada no bloquea el proceso.
- b) sigsuspend: Se debería reimplementar la llamada al Sistema para que el bloqueo del proceso se realizara mediante una espera activa.

4. Indica dos características que se mantienen y dos que se pierden cuando:

- a) Se crea un proceso hijo:
  - MANTIENE: imagen del proceso; reprogramación de signals
  - PIERDE: PID; signals pendientes de ser tratadas
- b) Un proceso cambia la imagen (ejecuta exec)
  - MANTIENE: PID; signals pendientes de ser tratadas
  - PIERDE: imagen del proceso; reprogramación de signals

5. ¿Qué pasaría si los procesos no tuvieran región de heap?

---

No se podría asignar/liberar memoria dinámica.

1. Explica el propósito del estado ZOMBIE.

Que un proceso pueda comunicar a su proceso padre su estado y razón de finalización mediante las llamadas `exit()` el hijo y `waitpid()` el padre, o que el padre pueda saber que el hijo ha acabado por un signal (y que signal) no capturado cuya acción por defecto sea acabar.