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

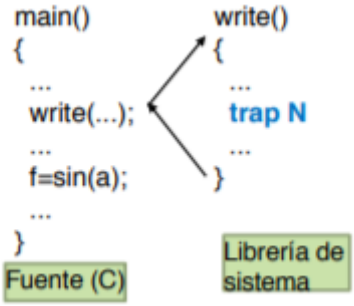
-Ejecución en modo privilegiado <- soporte HW

-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


main()
{
...
write(...);
...
f=sin(a);
...
}
Fuente (C)
write()
{
...
trap N
...
}
Librería de sistema

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.

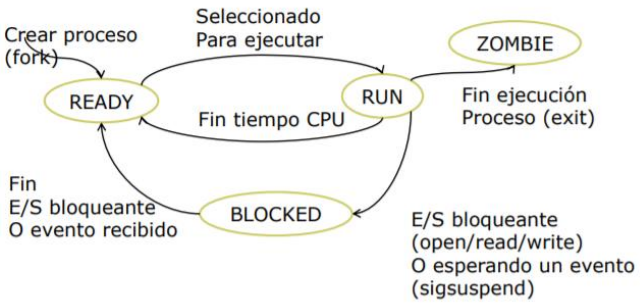
El **estado** suele gestionarse o con un campo en el PCB o teniendo diferentes listas o colas con los procesos en un estado concreto:

● **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
SIGCHLD	IGNORAR	Un proceso hijo ha terminado o ha sido parado
SIGCONT		Continua si estaba parado
SIGSTOP	STOP	Parar proceso
SIGINT	TERMINAR	Interrumpido desde el teclado (CtrC)
SIGALRM	TERMINAR	El contador definido por la llamada alarm ha terminado
SIGKILL	TERMINAR	Terminar el proceso
SIGSEGV	CORE	Referencia inválida a memória
SIGUSR1	TERMINAR	Definido por el usuario (proceso)
SIGUSR2	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 parámet

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

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

```
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