Resumen-SO.pdf

Llull179

Sistemas Operativos

2º Grado en Ingeniería Informática

Facultad de Informática de Barcelona (FIB)
Universidad Politécnica de Catalunya

Tema 1 – Introducción

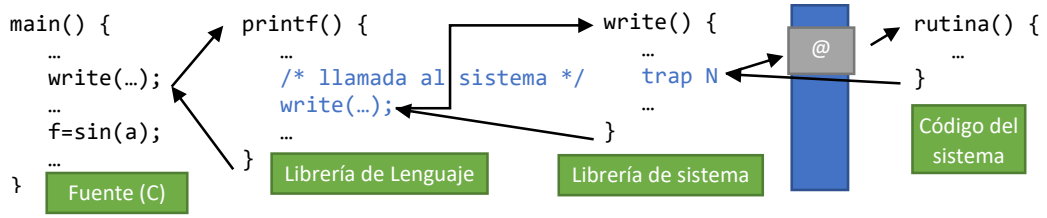
El SO hace de intermediario entre el usuario y el hardware, utilizando eficientemente los recursos disponibles y evitando que el usuario dañe estos mismos, de este modo el SO hace que sea para el usuario un entorno usable, seguro y eficiente.

2 modos (pueden haber más) de ejecución del SO:

- *User Mode* (NO-privilegiado)
- *Kernel mod*e (privilegiado): donde dejan ejecutar instrucciones privilegiadas, instrucciones que pueden llegar a dañar la máquina. Formas de acceder al código *kernel*:
  - **Interrupciones** (asíncronas) generadas por el hardware.
  - Los errores de software generan **excepciones** (síncronas).
  - Peticiones de servicio de programas: **Llamada a sistema** (síncronas).

Llamadas al sistema: se hacen desde el código (C, C++, …) se hacen desde el *User Mode*, se envían al *kernel* para su ejecución.

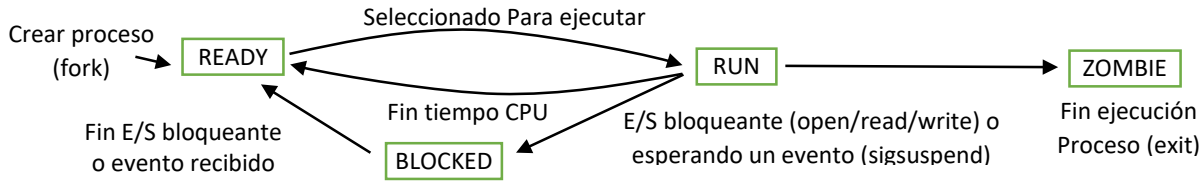
- Librerías "de sistema": contienen la parte compleja de la invocación, sirven para facilitar. (*User Mode*).
- Librerías "de lenguaje": Ofrecidas por los lenguajes de programación, ofrecen funciones de uso frecuente. (*User Mode*).



Tema 2 – Procesos

Conceptos:

- **Proceso**: representación del S.O. de un programa en ejecución. Este es nuevo cada vez que ejecutamos.
- **Programa ejecutable**: tiene código, datos y pila, inicializa los registros de la CPU, da acceso a dispositivos que necesitan acceso al modo *kernel*.
- **PCB** (*Process Control Bank*): gestiona la información de cada proceso. La PCB contiene 3 partes:
  - Espacio de direcciones: (1) de código, pila y datos
  - Contexto:
    - (2) Software: PID (*Process ID*, identificador único para cada proceso generado por el *kernel*), planificación, información sobre el uso de dispositivos, estadística, ...
    - (3) Hardware: tabla de páginas, *program counter*, ...
- **Paralelismo**: cuando realmente se ejecutan varios procesos a la vez (gracias a una arquitectura multiprocesador o *multi-core*).
- **Concurrencia** (paralelismo virtual): es la capacidad de ejecutar varios procesos de forma simultánea.
- **Procesos secuenciales**: independientemente de la arquitectura, los procesos se ejecutan uno detrás de otro.
- **Hilos de ejecución** (*Threads*): subproceso de ejecución y es la unidad mínima de planificación del SO, un proceso puede tener varios *threads*. Sirven para explotar el paralelismo, encapsular tareas… Son más versátiles que los procesos y al usar la misma memoria en un proceso pueden intercambiar información sin hacer llamadas al sistema.
- **Estados de un proceso**: para garantizar que todos los procesos se ejecutan y ninguno acapare todo el tiempo en la CPU.



Servicios básicos (UNIX)

`fork();`	Crea un proceso hijo en el punto en el que estaba el padre, se ejecutan de forma concurrente.
`waitpid(-1, NULL, 0);` (**Bloqueante**)	Espera a un hijo cualquiera.
`waitpid(pid_hijo, NULL, 0);` (**Bloqueante**)	Espera a un hijo a partir del PID.
`exit(0);`	Termina un proceso sin errores.
`exit(1);`	Termina el proceso con algún error
`execlp(const char *file, const char *arg, ...);`	Sirve para mutar = cambiar de ejecutable. Cambia el contenido del espacio pero mantiene el PID.
`getpid();`	Devuelve el PID del proceso hijo.
`getppid();`	Devuelve el PID del padre del proceso.

Bloqueante: es una llamada al sistema que puede forzar que deje el estado RUN (abandone la CPU) y pase a un estado en que no puede ejecutarse (WAITING)

Aspectos que el hijo **HEREDA**	Aspectos que el hijo **NO HEREDA**
Código, datos, pila, … La memoria física es nueva y contiene una copia de la del padre	PID, PPID
Programación de *signals*	Contadores internos
Máscara de *signals*	**Alarmas y *signals* pendientes**
Dispositivos virtuales	
userID y *groupID*	
variables de entorno	

Mientras el padre no haga `waitpid` no se libera el espacio que ocupa el PCB del hijo muerto (ESTADO ZOMBIE). Si el padre muere sin liberar los PCB's de sus hijos el sistema los libera (proceso init).

EJEMPLOS

FORK

```c
int ret = fork();
if (ret == 0) {
    // HIJO
} else if (ret < 0) {
    // ERROR
} else {
    // PADRE
    // ret == pid del hijo
}
```

ESQUEMA SECUENCIAL

```c
#define NUM_PROCESOS 2
int ret;
for (int i = 0; i < NUM_PROCESOS; i++) {
    ret = fork();

    if (ret < 0) control_error();

    else if (ret == 0) { // HIJO
        exit(0);
    }
    else { // PADRE
        waitpid(-1, NULL, 0);
    }
}
```

CREAR 2^n PROCESOS

```c
#define N 10

for(int i = 0; i < N; i++) {
    fork();
    printf("Hello World. I'm %d\n", getpid());
}
```

ESQUEMA CONCURRENTE

```c
#define NUM_PROCESOS 2
int ret;
for (int i = 0; i < NUM_PROCESOS; i++) {
    ret = fork();

    if (ret < 0) control_error();

     else if (ret == 0) { // HIJO
        exit(0);
    }
}

while (waitpid(-1, NULL, 0) > 0);
```

MIRAR EL EXIT_STATUS

```c
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

void tratar_exit_code(int status) {
    if (WIFEXITED(status)) {
        // Ha terminado por culpa de un exit
        int exitcode = WEXITSTATUS(status);
        printf("Ha terminado por un exit con exit_code: %d\n", exitcode);
    }

    else {
        // Ha terminado por un signal
        int signalcode = WTERMSIG(status);
        printf("Ha terminado con un signal con signal_code: %d\n", signalcode);
    }
}

int main(int argc, char const *argv[]) {
    for (int i = 0; i < 10; ++i) {

        int ret = fork();// FORK

        if (ret == 0) { // HIJO
            exit(i);
        }

        // PADRE
        int status;
        waitpid(-1, &status, 0);
        tratar_exit_code(status);
    }
    return 0;
}
```

Signals → Notificaciones que puede recibir un proceso (por el usuario o por el *kernel*) para informarle que ha sucedido con un evento (un evento es un *signal* asociado).

Hay dos *signals* que no están asociados a ningún evento SIGUSR1 y SIGUSR2.

Cada proceso tiene un **tratamiento** (que hacer cuando llegue la señal) por defecto asociado a cada *signal*, un tratamiento puede ser modificado excepto SIGKILL y SIGSTOP.

SIGNAL	TRATAMIENTO por defecto	EVENTO	Se pueden Bloquear/Desbloquear
SIGCHLD	IGNORAR	Un proceso hijo ha terminado o ha sido parado	SI
SIGCONT	-	Continua si estaba parado	SI
SIGSTOP	STOP	Parar proceso	No
SIGINT	TERMINAR	Interrumpido desde teclado (Ctrl + C)	SI
SIGALRM	TERMINAR	El contador definido por la llamada ha terminado	SI
SIGKILL	TERMINAR	Terminar proceso	No
SIGSEGV	CORE	Referencia inválida a memoria	No si son provocados por una excepción
SIGUSR1	TERMINAR	Definido por el usuario (proceso)	SI
SIGUSR2	TERMINAR	Definido por el usuario (proceso)	SI

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

- Una **máscara** es una estructura de datos que permite determinar qué *signals* (solo uno de cada tipo) puede recibir un proceso en un momento determinado de la ejecución.
- Cuando un proceso **bloquea** un *signal* el sistema lo marca como pendiente de tratar
- Cuando el proceso **desbloquea** el *signal* recibe el tratamiento

`kill(int pid, int signal);` → Enviar *signal*

```
sigaction(int signum, struct sigaction *tratamiento,
        struct sigaction *tratamiento_antiguo); →  Reprogramar un signal concreto
```

`struct sigaction`:

- sa_handler
  - SIG_IGN → Ignorar signal recibido
  - SIG_DFL → Tratamiento por defecto
  - my_func → función de usuario con una cabecera predefinida: `void nombre_funcion(int s);`
- sa_mask
  - Vacía → solo se añade el *signal* que se está capturando
  - Al salir se restaura la máscara anterior
- sa_flags
  - 0 → Configuración por defecto
  - 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.

`int sigprocmask(int operacion, sigset_t *mascara, sigset_t *vieja_mascara);` → Bloquear/Desbloquear *signals*.

- SIG_BLOCK → bloquea los *signals* de la máscara que le pases.
- SIG_UNBLOCK → desbloquea los *signals* de la máscara que le pases.
- SIG_SETMASK → intercambia las máscaras.

`int sigsuspend(sigset_t *mascara);` → Esperar hasta que llega un evento cualquiera (Bloqueante).

`int alarm(int num);` → Programa una alarma y devuelve el número de segundos restantes

Manipulación de máscaras con *signals*:

```
sigset_t mask;     // inicia máscara
sigemptyset(&mask);     // vacía
sigfillset(&mask);      // llena
sigaddset(&mask, SIGNUM)    // añade el signal a la máscara
sigdelset(&mask, SIGNUM)    // elimina el signal de la máscara
sigismember(&mask, SIGNUM)  // devuelve true si el signal está en la mascará, false si no.
```

Sincronización de procesos:

- Espera activa → Consume CPU ideal si vas a tardar poco.

```
void configurar_esperar_alarma() {
    alarma = 0;
}

void esperar_alarma() {  // Técnica del pesado
    while (alarma != 1); // Pregunta sin parar si ya se ha recibido el signal
}
```

- Espera pasiva (Bloqueo) → El proceso libera CPU

```
void configurar_esperar_alarma() {
    sigemptyset(&mask);
    sigaddset(&mask, SIGALRM);
    sigprocmask(SIG_BLOCK, &mask, NULL);
}

void esperar_alarma() {
    sigfillset(&mask);
    sigdelset(&mask, SIGALRM);
    sigsuspend(&mask);
}
```

Gestión interna de procesos → para gestionar procesos se necesita:

Estructuras de datos

- **PCB** → Para representar los datos de un proceso
  - PID
  - userID y groupID
  - Estado: RUN, READY, …
  - Espacio para salvar los registros de la CPU
  - Datos para gestionar *signals*
  - Información sobre la planificación
  - Información sobre la gestión de memoria
  - Información sobre la gestión de la E/S
  - Información sobre los recursos consumidos (*Accounting*)
- Gestionar y representar *threads* (depende del SO)

Estructuras de gestión → Que organicen los PCB's en función de su estado o de necesidades de organización del sistema.

- Cola de procesos → Todos los procesos creados en el sistema
- Cola de procesos en READY
- Cola esperando datos de algún dispositivo E/S

} El sistema mueve los procesos de una cola a otra según corresponda.

Algoritmo/s de planificación → Indica como gestionar las estructuras

- Política de planificación → Indica quien entra, durante cuánto tiempo, cuando se evalúa si hay que cambiar el proceso en la CPU y que pasa con el proceso que se estaba ejecutando, esto se ejecuta periódicamente. Las políticas de planificación son:
  - Preemptiva → La política le quita la CPU al proceso aunque este pudiera seguir ejecutándose.
  - No premptiva → La política no le quita la CPU él la "libera".

  Los procesos presentan ráfagas de computación y ráfagas de acceso a dispositivos E/S que lo bloquean
  - Procesos de cálculo → Consumen más tiempo haciendo cálculo que E/S.
  - Procesos E/S: consumen más tiempo haciendo E/S.

Mecanismos → Aplican las decisiones tomadas por el planificador

- Cambios de contexto (Context Switch) → Cuando un proceso deja la CPU y se pone otro proceso.
  - El sistema tiene que salvar el estado del proceso que deja la CPU y restaurar el estado del proceso que pasa a ejecutarse.
  - El cambio de contexto no es tiempo útil de la aplicación, así que hay que hacerlo rápido.
- Tiempo de ejecución de un proceso → Tiempo que pasa desde que llega al sistema hasta que termina.
- Tiempo de espera de un proceso → Tiempo que pasa en READY.

Ejecutando código usuario Proceso A	Modo usuario	int reloj
Salvar contexto Proceso A en PCB[A]	Modo *kernel*	
Planificador decide cambiar a proceso B		
Restaurar contexto Proceso B de PCB[B]		
Ejecutando código usuario Proceso B	Modo usuario	int reloj
Salvar contexto Proceso B en PCB[B]	Modo *kernel*	
Planificador decide cambiar a proceso A		
Restaurar contexto Proceso A de PCB[A]		
Ejecutando código usuario Proceso A	Modo usuario	int reloj

Round Robin (RR) → Los procesos se organizan según su estado. Están encolados por orden de llegada. El proceso recibe la CPU durante un quantum (10ms ó 100ms). El planificador hace una interrupción de reloj para que ningún proceso monopolice la CPU.

Eventos que activan la política Round Robin:

- Cuando el proceso se bloquea (no premptivo) → Pasa a la cola de bloqueados hasta que termina el acceso al dispositivo.
- Cuando el proceso termina (no premptivo) → El proceso pasa a zombie en el caso de Linux o terminaría.
- Cuando termina el quantum (premptivo) → El proceso se añade al final de la cola de READY.

Ningún proceso espera más de (N - 1) * Q milisegundos. Donde N es el número de procesos y Q el tiempo de quantum.

- Q grande → es como si fuesen orden secuencial
- Q pequeño → produce overhead si no es muy grande comparado con el cambio de contexto.

¿Qué hace el kernel cuando se ejecuta un…?

- Fork
  - Busca PCB libre y lo reserva.
  - Inicializar datos (PID, ...).
  - Se aplica política de **Gestión de memoria.**
  - Se actualizan las estructuras de **Gestión de E/S.**
  - En el caso de RR → Se añade a la cola de READY.
- Exec
  - Se substituye el espacio de direcciones por el código/datos/pila del nuevo ejecutable
  - Se inicializan las tablas de *signals*, contexto, …
  - Se actualizan las variables de entorno, argv, registros, …
- Exit
  - Se liberan todos los recursos del proceso.
  - En Linux se guarda el estado de finalización en el PCB y se elimina de la cola de READY.
  - Se aplica la política de planificación.
- Waitpid
  - Se busca el proceso en la lista de PCB's para conseguir su estado de finalización.
  - Si el proceso estaba zombie, el PCB se libera y se devuelve el estado de finalización a su padre.
  - Si no estaba zombie, el proceso padre se elimina pasa de estado run a bloqued hasta que el proceso hijo termine y se aplicaría la política de planificación.

Protección y seguridad

- Físico → Poner las máquinas en habitaciones / edificios seguros.
- Humanos → Controlar quien accede al sistema
- SO
  - Evitar que un proceso sature el sistema
  - Asegurar que determinados servicios funcionen
  - Asegurar que ciertos puertos de acceso no están operativos
  - Controlar que los procesos no se salgan de su espacio de direcciones.
- RED → Es el más atacado