Teoria 2 - Llamadas al Sistema

Implementación de System Calls

A nivel de procesador, la llamada al sistema ocurre mediante una instrucción especial del procesador (syscall, int, trap, etc). Esa instrucción cambia a modo privilegiado.
Al nivel del programador, la llamada es una subrutina que escribimos en el código fuente. El compilador la sustituirá por una invocación a la instrucción especial, con los argumentos que sean necesarios.

Ejemplos
Windows

```
handle = OpenFile("mifichero",ofstruct,OF_READ);
```

UNIX

```
fd = open("mifichero",O_RDONLY);
```

MS-DOS

```
MOV AH,3DH
MOV AL,0
MOV DX,StringMiFichero
INT 21h
```

Pasaje de Argumentos

Los parámetros pueden pasarse mediante registros de CPU, escribiendolos en una tabla en memoria principal o colocándolos en la pila.

En C, la biblioteca estándar `stdio.h` no pertenece al SO y se ejecuta en modo usuario. Por ejemplo, `printf()` utiliza la llamada al sistema `write()` para poder escribir en la consola.

Mecanismos de interrupciones

Se pueden clasificar en tres tipos:

- Llamadas al sistema (traps): son provocadas por el programador. Existen distintas instrucciones para provocar una interrupción al ejecutar una de ellas (INT, syscall, etc).
- Excepciones: se ejecutan de forma síncrona al procesador, por lo que pueden predecirse analizando la traza del programa que estaba ejecutándose en ese momento en la CPU. Las causas para estas suelen ser operaciones no permitidas, como división por 0, overflows, accesos a memoria no permitida, etc.

- Interrupciones (de dispositivos o hardware): son asíncronas al procesador y externas. Pueden producirse en cualquier momento y las causas son externas al procesador, a menudo ligadas con dispositivos de E/S.

Procesos

Es una abstracción del SO para identificar un programa en ejecución. Para ejecutarse necesita recursos (memoria, dispositivos de E/S, CPU, etc) y suele usar el segmento text (código), data (variables globales y memoria dinámica) y la pila (datos locales de subrutinas).

Estados de un proceso

El SO administra los procesos asignándoles estados

- Nuevo: recién creado por el SO
- Ejecutando: está en la CPU ejecutando instrucciones
- Bloqueado: esperando a que ocurra algún evento (terminación de una operación de E/S o la recdepción de una señal)
- Listo: esperando a que le asignen un procesador
- Terminado: no ejecutará más instrucciones y el SO le retirará los recursos que consume

Transiciones:

- null -> nuevo: se crea un nuevo proceso para ejecutar un programa.
- nuevo -> listo: el proceso está listo para ejecutarse (que no haya demasiados procesos activos y que no se degrade el rendimiento del sistema). 1
- listo -> ejecutando: el planificador selecciona un nuevo proceso para ser ejecutado de entre los que están en estado "listo".
- ejecutando -> terminado: el proceso termina por parte del SO, porque haya completado su ejecución o porque se haya abortado por un error.
- ejecutando -> listo: el proceso alcanzó el máximo de tiempo ininterrumpido posible en ejecución, fue expulsado para ejecutar otro de mayor prioridad o por auditorías o mantenimiento. 2
- ejecutando -> bloqueado: el proceso solicita un recurso por el que debe esperar o realizo una system call que debe ser atendida para continuar.
- bloqueado -> listo: el evento por el que estaba esperando es completado.
- listo -> terminado: el proceso termina porque el padre finalizó o el padre lo finalizó
- bloqueado -> terminado: idem.

Contexto de un Proceso

Es la información relativa al proceso en un instante de tiempo determinado. Esta información debe ser conocida por el núcleo para iniciar o continuar su ejecución. Cuando el núcleo pasa a ejecutar otro

proceso, debe cambiar el contexto, ya que debe ejecutar un proceso en el contexto del mismo.
El contexto está constituido por el espacio de direcciones virtuales del mismo, el contenido de los registros de la máquina y las estructuras de datos del núcleo asociadas al mismo.
Se puede separar en

- Contexto a nivel de usuario: código, datos, pila de usuario, memoria compartida que ocupan el espacio de direcciones virtuales del proceso.
- Contexto de registros: contador de programa, registro de estado (modo, prioridad, overflow, etc), puntero de pila, registros de propósito general, etc.
- Contexto a nivel de sistema: tabla PCB (Process Controll Block), pila del núcleo, tabla de páginas, etc.

Además, en un contexto puede distinguirse una parte *dinámica*, que debe guardarse ante ciertos eventos, y una parte *estática*, que no es necesario guardar. La parte dinámica está formada por el contexto a nivel de registros y su pila del núcleo, mientras que el resto son su parte estática.

Cambio de contexto

Al comenzar o continuar la ejecución de un proceso, el SO debe aplazar o finalizar la ejecución de otro. Esto requiere un conjunto de tareas para **cambiar de contexto**.
Los cambios de contexto pueden deberse a:

- Entrada de un proceso bloqueado.
- Finalización de la ejecución con una system call *exit()*.
- Vuelta al modo usuario luego de atender una interrupción y encontrar un proceso con mayor prioridad.
- Finalización del tiempo de uso del procesador por parte de un proceso.

Algorítmo de cambio de contexto:

- Decidir si hay que cambiar de contexto, según las situaciones planteadas anteriormente.
- **Guardar** el contexto del proceso actual.
- Según el argorítmo de planificación, elegir el próximo proceso que va iniciar o continuar su ejecución.
- Restaurar contexto del proceso elegido.

Bloque de Control de Proceso (PCB)

El SO mantiene la información sobre los procesos en una **tabla de procesos**. Cada entrada de esta tabla es llamada "Bloque de Control de Procesos" (PCB) y mantiene *casi* toda la informacón sobre un proceso [3]:

- Identificación:
  - PID

    - PID del padre
    - User ID
- Estado
    - Estado del proceso
    - Evento que espera, si está bloqueado
    - Prioridad
    - Memoria asignada
    - Archivos abiertos
    - Puertos usados
    - Temporizadores
- Control
    - Estado del procesador

Creación de Procesos

En UNIX la creación de un proceso se realiza por la llamada a sistema *fork()* en modo kernol dentro de un proceso padre. Esta llamada realiza las siguientes acciones:

- Solicita la entrada en la tabla de procesos
- Asigna un identificador
- Copia una imagen del proceso padre, exceptuando la memoria compartida.
- Incrementa el contador de los ficheros del padre, reflejando el proceso adicional que ahora los posee [4]
- Asigna el estado de Listo
- Devuelve el identificador al padre, mientras que devuelve 0 al hijo. Esto sire para distinguir en ejecución

Luego de estas acciones pueden suceder una de estas situaciones:

- Continuar con el proceso padre, volviendo a modo usuario
- Trasferir el control al hijo, que comienza en el mismo punto de retorno de la llamada fork.
- Transferir el control a otro proceso.

Wait()

La función `wait()` espera que alguno de los hijos termine su ejecución para continuar la ejecución actual. Ver documentación
La función `waitpid(pid_t)` espera a que termine la ejecución de un proceso hijo especificado por el argumento.

Exec()

Las funciones `exec` reemplazan la imágen [5] del proceso actual con una nueva. El código, datos, bss y la pila se reescriben con los del programa ejecutado.
`execl`, `execlp`, `execle`, `execv`, `execvp` son interfaces de `exeve`. Ver documentación.

Consultas

Consulta 1

¿Qué sería un proceso que degrade el rendimiento del sistema y cómo lo detectaría de antemano el SO?

Consulta 2

¿A qué le llamaríamo auditoría o mantenimiento? ¿El usuario lo detuvo para debug por ejemplo?

Consulta 3

¿Cómo que casi?

Consulta 4

¿A qué contador de los ficheros se refiere?

Consulta 5

¿Qué sería la imagen del proceso?