Teoria 1 - Introducción

¿Qué es un SO?

Existen varias definiciones según el autor. Según Silberschatz, es **un** programa intermediarío entre el usuario y el hardware. Según Tanenbaum, es una capa de software que administra los recursos y sirve de interfaz entre los programas de usuario y el hardware. Según Deitel & Deitel, es una **serie de programa** en software o en la memoria fija que hacer utilizable el hardware, poniendo a disposición del usuario el "poder computacional basico" del hardware. Según Milenkovic, es una colección organizada de extensiones software del hardware, que consiste en rutinas de control y proporciona un eterno para la ejecución de los programas. Según Pérez Campanero, es un **conjunto de programas** que se relacionan ordenadamente entre sí para contribuir a que el ordenador lleve adelante el trabajo que se le encomienda.

Extensión de la máquina

Un SO presenta una interfaz para el usuario con el hardware que puede tomar la forma de línea de comandos, proceso por lotes o una interfaz gráfica de usuario (GUI). Para esto crea abstracciones como los archivos, procesos, memoria, etc., ocultando las características del hardware como interrupciones, heap, stack, temporizadores y otra funciones de bajo nivel.

Funciones de un SO

Administrar Hardware

El SO se encarga de

- Gestionar recursos: la CPU, memoria principal, almacenamiento secundario y periféricos de entrada y salida.
- Gestor de información o archivos: controla la creación, borrado, lectura y escritura de archivos y programas.
- Administración de tareas: la información de los programas y procesos ejecutándose en la computadora. Puede cambiar la prioridad de estos procesos, terminarlos y el uso que hacen de la CPU.
- Soporte: los *servicios de soportes* de cada SO dependen de sus implementaciones. Puede tratarse de inclusión de utilidades, actualización de versiones, seguridad, controladores nuevos de periféricos o corrección de errores de software.

Proteger Hardware

Para que la computadora funcione, hay que impedir que los programas de usuario realicen ciertas operaciones libremente, como pueden ser

- Acceder a memoria del SO y otro programas
- Acceder directamente a los dispositivos de E/S
- Utilizar libremente la CPU todo el tiempo que quieran.

Para esto se introducen diferentes niveles de privilegios. Esto se logra con un **modo dual de operación** o privilegios.
Existen entonces

- Modo usuario: ejecuta aplicaciones, posee un repertorio limitado de registros e instrucciones y se imponen limitaciones en el direccionamiento de memoria.
- Modo kernel: tiene acceso total al hardware y puede acceder a todas las instrucciones, registros y memoria.
  Ejemplo: en los procesadores x86 existen cuatro niveles de proteccion, o jerarquías.
- Anillo 0: tiene los mayores privilegios (modo kernel)
- Anillo 1 y 2: se utilizar para los controladores de dispositivos.
- Anillo 3: el nivel de privilegio más bajo (modo usuario)
  En Linux, Windows XP y anteriores solo se utilizan el anillo 0 y 3.

Arquitecturas

Arquitectura Monolíticas

Todo el SO se ejecuta como un solo programa en modo kernel. Se escribe como una colecciones de procedimientos que pueden comunicarse entre sí sin restricciones. Ejemplo de estos son MS-DOS, UNIX, MaxOS, Windows 9x, etc. Para cada llamada al sistema existe un procedimiento de servicio que atiende la llamada y ejecuta los procedimientos unitarios, que pueden necesitar varios procedimientos de servicio, como obtener datos de los programas de usuario.

Ventajas:

- Comunicación rápida, limpia y homogenea entre módulos mediante llamadas a procedimientos.

Desventajas:

- Baja escalabilidad:
  - Dificil de entender
  - Dificil de modificar
  - Dificil de mantener
- Baja confiabilidad:

- No existe aislación entre los componentes

Arquitectura Microkernel

Divide el SO en módulos pequeños y bien definidos, de los cuales solo el *microkernel* se ejecuta en modo kernel y el resto se ejecuta en modo usuario. Las funciones centrales son manejadas por el kernel y la interfaz del usuario es manejada por el entorno (shell). La planificación de los hilos para multitareas recae en el microkernel.
Ejemplos de estas son AIX, BeOS, Mach, Minix, Symbian, L4, K42, etc.

Ventajas:

- Gran confiabilidad
- Escalabilidad
- Portabilidad
- Concurrencia
- Flexibilidad
- Confiabilidad
- Seguridad
- Simplicidad

Desventajas:

- Bajo rendimiento: la comunicación a través de mensajes no es eficiente
- Complejidad: es complicado sincronizar los módulos

Arquitectura de Capaz o Jerarquía

Organiza el SO como en capas con una jerarquía. Cada capa se construye sobre otra, es decir que la capa `n` puede utilizar los servicios de la capa `n-1`, que ofrece una interfaz clara y bien definida y a su vez solo utiliza los servicios que le ofrece la capa inferior.

Ventajas:

- Modularidad
- Ocultación de información

Desventajas:

- Poco flexible
- Bajo desempeño

Arquitectura por módulos o híbridos

El kernel se compone por módulos separados de forma independiente, de forma que si uno falla, no afecta al resto ni al núcleo. El resto de operaciones siguen sus funciones habituales. Se parece a la de capas, pero es mucho más flexible, ya que cualquier módulo puede llamar a otro. También se puede paracer a la microkernel, excepto que incluye código adicional en espacio de kernel para aumentar el rendimiento.

Llamadas al Sistema

Los programas invocan los servicios del SO por medio de las **system call**.

- Siguen el estándar POSIX (Portable Operating System Interface, y X de UNIX) en UNIX y en Linux.
- Win32 en Windows.
- Sirven de interfaz entre las aplicaciones y el SO.
- Se invocan con un trap o interrupción de software
- Se acceden como función en los lenguajes (C, C++, etc).

El número y tipo de llamadas varia según el SO. Por lo general, existen llamadas para ejecutar ficheros que contienen programas, pedir más memoria dinámica para un programa, realizar tareas de E/S, crear procesos, directorios, etc.

Ejemplo: en C las funciones de `stdio.h` utilizan otras de más bajo nivel que son llamadas *system call wrappers* que hacen la llamada a las system call reales.

Este código

```c
#include <stdio.h>
int main(void) {
    printf("Hello world!\n");
    return 0;
}
```

Es equivalente a este

```c
#include <unistd.h>
int main(void) {
    const char *msg = "Hellow world!\n";
    write(STDOUT_FILENO, msg, 13);
    return 0;
}
```

También se puede llamar directamente a las system call en el kernel utilizando ensamblador:

```
section .data
    msg db "Hello world!", 0xa, 0x0d
```

```
    largo equ $-msg

section .text

global inicio
inicio:
    mov eax, 4          ; num de llamada al sistema (sys_write)
    mov ebx, 1          ; file handle (stdout)
    mov ecx, msg        ; puntero a la cadena
    mov edx, largo      ; longitud de la cadena
    int 80h                   ; llamada al kernel

    mov eax, 1          ; num de llamada al sistema (sys_exit)
    xor ebx, ebx        ; codigo de salida 0 (exitoso)
    int 0x80            ; llamada al kernel
```

Este codigo puede probarse instalando `nasm` y `ld` (binutils). Luego ensamblamos y enlazamos con

```
nasm -f elf holamundo.asm
ld -o holamundo holamundo.o
```