Introducción

Un sistema de computación está formado por múltiples componentes de *hardware* y *software*. La única manera de lidiar con la complejidad de estos sistemas es *modularizar*, es decir, crear pequeños componentes interconectados para construir sistemas más complejos. La siguiente figura muestra una estructura típica de componentes en diferentes capas de abstracción un sistema de computación actual de propósitos generales.

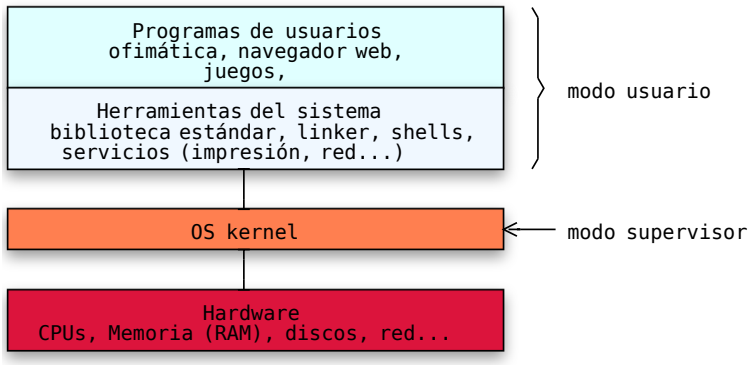


Figura 1: *Capas de un sistema de computación.*

Para comprender mejor el rol que cumple un *sistema operativo* es conveniente analizar la historia del desarrollo y la evolución de sistemas de computación.

Los primeros sistemas de cómputo no tenían *interfaces* para el usuario/ programador. Los códigos de las instrucciones y los valores de los datos de un programa se debían introducir manualmente en la memoria principal del sistema por medio de cableados y conmutadores.

Obviamente esto era una tarea sumamente tediosa y propensa a errores y sólo era posible hacer pequeños programas para las pequeñas memorias de aquellos días.

A medida que la capacidad de memoria crecía, se desarrollaron dispositivos de almacenamiento para los programas y datos de entrada y salida:

1. Tarjetas y cintas de papel perforadas
2. Medios magnéticos como las cintas, cilindros y discos
3. Actualmente: Memorias no volátiles de estado sólido

Para el caso de las cintas y tarjetas perforadas de papel, se desarrollaron dispositivos que comúnmente eran computadoras de propósitos específicos o *teletipos (tty)* adaptados que permitían que los programadores *escriban* el programa en la *teletipo* (una máquina de escribir electrónica) y ésta producía las cintas o tarjetas perforadas. La computadora incluía un *lector* de tarjetas o cintas que introducía las instrucciones y datos de los programas en la memoria y un *impresor* que perforaba en las tarjetas o cintas los datos de salida.

En esos días (principios de 1950) no existía el concepto de sistema operativo. Cada programa debía ser *auto-contenido*, o sea que debía contener todas las instrucciones necesarias aún para las operaciones rutinarias de entrada-salida, por lo que cada programador debía conocer todos los detalles del sistema.

Actualmente podemos encontrar sistemas sin SO comúnmente en pequeños *micro-controladores* o *digital signal processors (DSPs)* en donde la única aplicación que ejecutan es un programa monolítico que contiene todas las funciones necesarias. Estos sistemas implementan un mini sistema de cómputo de propósitos específicos.

Algunos SOs para pequeños procesadores tienen la forma de una *biblioteca* que se enlaza con el código de la aplicación. Un ejemplo es el *Real-Time Operating Systems (RTOS)* usados para el desarrollo de *embedded systems*. Esto es similar a los primeros sistemas operativos de la década de 1950.

Los primeros sistemas operativos

Obviamente, los programadores debían escribir mucho código que podría ser *reusable* por todos. En particular las operaciones de entrada-salida y otras funciones de uso común o *rutinario* (de ahí el nombre *routines*) como funciones matemáticas y otras.

Los sistemas denominados ***por lotes*** o ***batch systems*** permitían cargar programas y *bibliotecas* (contenedores de rutinas). Los programadores depositaban sus programas y datos de entrada(*jobs*). Luego un operador los agrupaba (en lotes) y los ordenaba en una cola para que el sistema los cargara y ejecutara en secuencia. Cada lote era precedido por una secuencia de comandos que especificaba las *bibliotecas* requeridas. El sistema operativo (SO) las cargaba en memoria si no habían sido cargadas previamente.

En estos tipos de sistemas cada programa tenía acceso a todos los recursos (CPU, memoria, dispositivos de entrada-salida, etc) del sistema.

Sistemas operativos multitarea

Las operaciones de entrada-salida dejan demasiado tiempo a la CPU ociosa, la cual sólo debía esperar (en un ciclo, verificando algún registro del controlador del dispositivo) a que esas operaciones finalicen.

Con el objetivo de maximizar el uso de CPU, los próximos sistemas operativos evolucionaron para cargar en memoria varios programas (*procesos*) y ejecutarlos concurrentemente asignando la CPU a otra tarea cuando la anterior iniciaba una operación de entrada-salida. Estos sistemas se denominan ***sistemas de multiprogramación*** o *multitareas*.

De esta manera se logra que se utilice la cpu y las operaciones de E/S en paralelo.

Generalmente los controladores de dispositivos generan interrupciones al ocurrir un evento como la finalización de una operación de lectura o escritura de un bloque en el disco.

Ante una interrupción recibida por la CPU, ésta hace que el flujo de ejecución *salte* a una *rutina de atención* o *trap handler*. De ese modo, el kernel retoma el control.

En estos sistemas, las tareas *cooperan* cediendo la CPU en cada llamada al sistema por lo que se conocen como *SO multitarea cooperativa*. Las primeras versiones de MS-Windows tenían esta característica.

Cambios de contexto

Cuando el kernel de un SO decide quitar la CPU a un proceso y asignarla a otro se realiza un *context switch* y consiste en los siguientes pasos:

1. Guardar (salvar) el estado (*registros*) de la CPU en el descriptor del proceso.
2. Recuperar en los registros de la CPU el estado guardado del próximo proceso.

Esto producirá un *salto* de un punto de ejecución de un hilo de ejecución al otro ya que cambian el *program counter (pc)* y el *stack pointer (sp)*.

En el capítulo de gestión de procesos se describe este mecanismo en mayor detalle.

Protección

Estos sistemas requieren de algún mecanismos de *protección*:

- **Confinamiento**: Cada proceso debe acceder *sólo a su propio espacio de memoria*.
- **Control**: Los procesos no deberían poder ejecutar ciertas *instrucciones privilegiadas*, como deshabilitar interrupciones o usar registros especiales que permitan quitarle el control al kernel.

El kernel es un binario independiente y auto-contenido. Comúnmente tiene acceso a todos los recursos del sistema.

Cada aplicación en ejecución (*proceso*) tendrá su *propio espacio de direcciones de memoria*.

La protección de memoria requiere de un mecanismo especial para que una tarea requiera un servicio del SO ya que no puede simplemente hacer una llamada a una función del kernel (ya que son programas diferentes).

Es necesario que el hardware disponga de un mecanismo para acceder a los servicios del SO como por ejemplo, una instrucción `trap` (trampa).

> **Una *Llamada al sistema (syscall)* es una función que permite a los procesos solicitar un servicio o recurso al kernel del SO. La *interfaz de un OS* comúnmente se describe como el conjunto de *syscalls* que ofrece a las aplicaciones.**

En una *syscall* el flujo de ejecución del proceso se interrumpe *saltando* del espacio de memoria de un proceso al espacio del kernel. Luego, el kernel procesa el syscall y retorna al espacio de usuario del proceso el cual continúa su ejecución.

Durante el procesamiento de una syscall el kernel puede decidir quitarle la CPU al proceso corriente (porque ya) y continuar la ejecución de otro.

Generalmente, en un salto al kernel, la CPU cambia su ***nivel de privilegio*** o ***modo de operación***. La mayoría de las CPU modernas incluyen al menos dos modos de ejecución: ***modo kernel (o supervisor)*** y ***modo usuario***.

> **La CPU ejecuta el código del núcleo del SO en *modo kernel* o *supervisor* que es más privilegiado que el modo usuario en el cual ejecutan los *procesos de usuario*. En modo kernel la CPU permite la ejecución de *instrucciones privilegiadas* y acceder a registros de dispositivos, habilitar y deshabilitar interrupciones, entre otras. Estas instrucciones privilegiadas no son accesibles en *modo usuario*.**

Sistemas de tiempo compartido

A medida que se incorporaron dispositivos interactivos como las *consolas* (teclado + pantalla), se comenzaron a desarrollar aplicaciones *interactivas*, las cuales permitían a usuarios a ingresar datos y visualizar resultados en una *sesión* de ejecución de un programa.

En un sistema con multi-programación una tarea puede monopolizar la CPU o hacer que el sistema sea muy poco interactivo ante usuarios (éstos podrían tener demoras significativas si la tarea en ejecución no hace una llamada al sistema).

Para evitar este problema el kernel puede *quitarle (preempt) la CPU* a una tarea que ya la ha usado por un intervalo de tiempo determinado y asignársela a otra tarea y garantizar un progreso más uniforme. Esto se logra mediante el manejo de las *interrupciones periódicas de un reloj*, las cuales son ***atrapadas*** por el *kernel* y actuando en consecuencia.

Los sistemas *multi-usuarios* soportan diferentes espacios de trabajo para diferentes usuarios (*home directories*) con sistemas de control de acceso a los recursos. Muchos de estos sistemas permiten la conexión de múltiples *terminales* o *consolas* a la computadora la cual se usa de manera compartida.

Actualmente las computadoras personales permiten *múltiples consolas virtuales*. Una *consola* o *terminal* virtual es un programa que *emula* una terminal física (hardware). También existen aplicaciones que *emulan* terminales remotas por red como telnet, ssh y aplicaciones de *remote desktop*.

Objetivos de un sistema operativo

Un SO tiene varios objetivos. El principal es ofrecer una ***máquina abstracta*** que oculta los detalles del hardware.

Uno de los principales objetivos es *virtualizar* recursos de hardware.

Esta *virtualización* permite su mejor utilización ya que incluye el *multiplexado* de los mismos.

Básicamente un SO crea ***abstracciones*** de los recursos.

Por ejemplo, un *proceso* representa una unidad de ejecución en una CPU (virtual) y su memoria. Un SO multitarea *multiplexa* la CPU entre los procesos.

En algunos casos como en la gestión de la memoria, ésta se *divide* lógica o físicamente para asignarle espacios a cada proceso y al kernel. Un sistema de memoria virtual permite ejecutar un conjunto de procesos cuya demanda de memoria es mayor que la memoria física correspondiente mediante la técnica de *swapping* (que se describe en el capítulo [memory.md]).

Otro ejemplo de *abstracción* del hardware es el ***sistema de archivos*** el cual ofrece a los procesos una *interfaz* de operaciones sobre *archivos* y *directorios (carpetas)*, ocultando los detalles técnicos de los dispositivos de almacenamiento (discos, cintas, etc).

El diseño de UNIX abstrae la interacción con los dispositivos de E/S presentando a éstos como *archivos especiales* que se pueden leer y/o escribir mediante la conocida API de llamadas al sistema sobre archivos como `open(...)`, `read(...)`, `write(...)`, `close(...)` y otras.

Esta *virtualización* o *multiplexado* hace que el SO también sea sea un ***administrador de recursos*** ya que debe incluir algoritmos de planificación de la asignación de los recursos para optimizar su uso.

Hardware

El siguiente diagrama muestra un esquema simplificado de una arquitectura general del hardware.

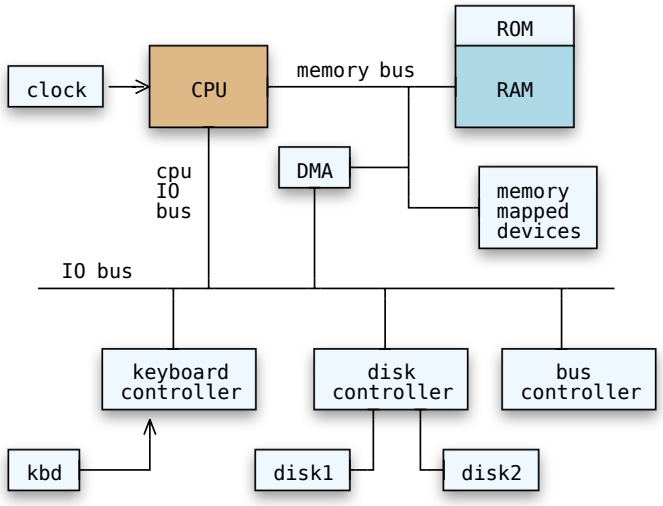


Figura 2: *Estructura típica de hardware.*

Como se puede apreciar en la figura de arriba, el hardware de un sistema de computación consta de diferentes componentes.

- *CPU*: Uno o más procesadores.
- *ROM*: Memoria *no volátil* que contiene código de arranque (boot) y datos de la plataforma de hardware.
- *RAM*: Memoria principal (*random-access-memory*). Esta memoria es *volátil*.
- *I/O bus and ports*: Bus de I/O al cual se conectan los dispositivos de entrada/salida como en la plataforma x86 de Intel.
- *Device controller*: expone un conjunto de *puertos* o *registros* de control y estado para un cierto tipo de dispositivos. Un *controller* puede manejar varios dispositivos del mismo tipo. Existen varios estándares como **IDE/PATA** para dispositivos de almacenamiento, **USB** para varios tipos de dispositivos.
- *Memory mapped devices (MMIO)*: Dispositivos de entrada salida conectados al *memory bus*. Ejemplos: *display/video controllers* y otros.
- *Direct-Memory-Access (DMA)*: Dispositivo que inter-conecta controladores de dispositivos al bus de memoria para permitir la transferencia de datos desde/hacia los dispositivos sin necesidad que intervenga la CPU.

Un SO debe conocer los detalles de estos componentes. Los módulos de software que interactúan con los controladores de dispositivos se denominan *device drivers*.

Diseño

Los SO están diseñados en forma modular, en subsistemas:

1. Núcleo (kernel):
    - Gestión de *procesos* y usuarios
    - Gestión de la *memoria*
    - Subsistema de *entrada-salida*: Conjunto de *device drivers*
    - Sistemas de *archivos*
    - Subsistema de *red* (comunicaciones)
    - Mecanismos de *Seguridad*: Generalmente es un sistema transversal a los demás subsistemas
    Aca falto el IPC, comunicacion entre procesos.
2. Aplicaciones y *bibliotecas* del sistema (ej: `libc` en sistemas tipo UNIX)
    - *Utilidades*: Shells, comandos básicos, editores de texto, ...

- *Servicios*: Procesos de usuario (*daemons*) que brindan servicios de uso común como el servicio de impresión u otros. Red Tambien es un servicio que tienen los sistemas.
  Tambien bluetooth, servicio de audio, etc

Algunos de las procesos de usuario que forman parte de las utilidades y servicios del sistema ejecutan con mayores *privilegios* que los programas de usuario comunes.

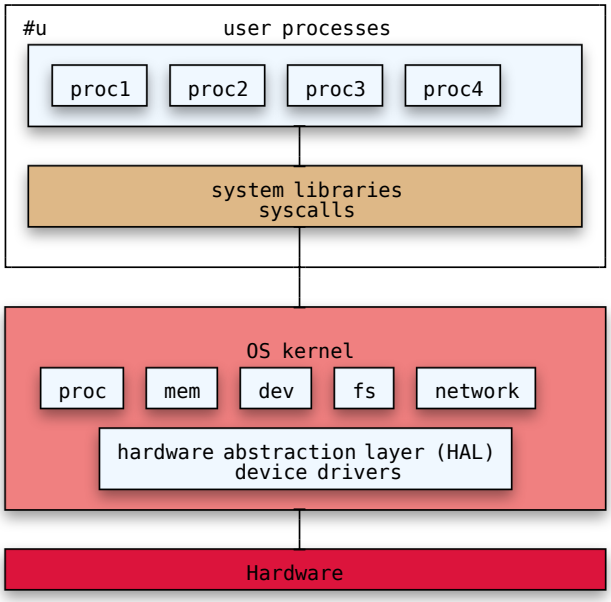


Figura 3: *Arquitectura de un sistema de computación.*

En la figura 3, los programas de usuario utilizan (*se enlazan con*) las bibliotecas del sistema y otras de propósitos específicas. La biblioteca del sistema contiene la *interfaz del programador de aplicaciones (API)* la cual incluye las funciones que implementan las *llamadas al sistema* y otras utilidades de uso rutinario.

Una *llamada al sistema (syscall)* activa el mecanismo de interrupciones para *saltar* al código del kernel que la procesa.

El subsistema de E/S, representado como `dev` en la figura 2, incluye los *device drivers* los cuales son módulos de software que *controlan* una cierta clase de dispositivos, como discos IDE, teclado, mouse, pantalla, etc.

Uno de los desafíos en el diseño de un SO es encapsular el software dependiente de la plataforma de hardware del resto del código independiente de la plataforma. Esto se logra definiendo abstracciones de los componentes de hardware para facilitar su portabilidad a otras arquitecturas.

Esta capa de abstracción en el *kernel* se conoce como *hardware abstraction layer (HAL)* y se considera la capa más baja dentro del kernel.

Los diseños de los SO se puede clasificar en:

- ***Monolíticos***: Los subsistemas del *kernel* se *enlazan* en un solo *ejecutable*. Cada módulo puede *invocar* a funciones de otros módulos.

  - Ventajas: Simple y eficiente.
  - Desventajas: Un error en un módulo puede afectar a todo el núcleo.

  Ejemplos: UNIX, [GNU-Linux][**https://www.gnu.org/gnu/linux-and-gnu.en.html**], **FreeBSD** y **Fuchsia**.

- ***Microkernels***: Cada subsistema del *kernel* se compila como un *ejecutable* independiente. Para *invocar* un servicio de otro subsistema se usa un mecanismo de comunicación basado en *mensajes*.

- Ventajas: Un error (ejemplo, de memoria) en un subsistema no afecta a los demás ya que cada componente es independiente y está confinado en su propia área de memoria. Fácilmente se puede convertir en un SO *distribuido' (subsistemas ejecutándose en diferentes nodos de una red).
- Desventajas: El mecanismo de pasaje de mensajes crea una *sobrecarga* computacional importante afectando su rendimiento.

Ejemplos: **Minix**, **L4**, **MS-Windows**

- ***Máquinas virtuales***: El sistema tiene un ***hypervisor*** que se encarga de ***multiplexar*** el hardware a los SO *huéspedes* que se ejecutan encima (en modo usuario). El *hypervisor* ofrece a cada SO huésped un *ambiente aislado* de ejecución.

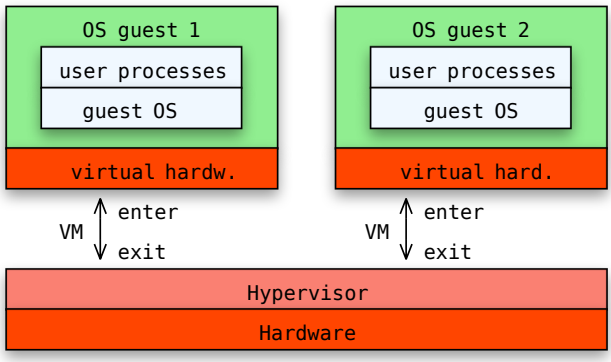


Figura 4: *Esquema de un SO basado en máquinas virtuales.*

Cuando el *hypervisor* está integrado en un SO, a éste se le denomina el *host kernel (anfitrión)* como por ejemplo **Linux KVM** que provee mecanismos básicos para las *transiciones* entre un SO *huésped (guest)* y el *host* y redirigir a un virtualizador en modo usuario como **qemu** o **VirtualBox**.

Generalmente para poder implementar un hypervisor, el hardware debe proveer mecanismos como nuevos *modos de operación* de la CPU como por ejemplo *Intel VMX root mode* (modo del *anfitrión*) y *VMX non-root-mode* (modo del *huésped*) y otras estructuras de datos como *registros sombras (shadowed)*.

Algunas arquitecturas modernas como RISC-V incluyen modos de protección o *privilegios* para facilitar la implementación de la virtualización.

Ejemplos: **Xen** e IBM System 370.

> ! No se debe confundir la virtualización con *contenedores*. Un contenedor es un *grupo de procesos* y recursos (memoria, filesystem, dispositivos y otros) que son independientes del resto del sistema. Esto permite la creación de *sistemas auto-contenidos minimalistas*, es decir que incluyen sólo las dependencias requeridas. Se ejecutan en forma *confinada o aislada* del resto del sistema. El SO debe proveer mecanismos para su ejecución. Ejemplos: GNU-Linux *cgroups* y FreeBSD *jails*. Aplicaciones como **Docker** permiten crear, configurar y ejecutar contenedores.

Consideraciones de diseño

Muchos SO actuales tienen una arquitectura híbrida, generalmente con un kernel pequeño monolítico y otros componentes *fuera de él* comunicándose por mensajes. Ejemplos de estos diseños son MS-Windows, Mac-OS y FreeBSD.

Algunos SO modernos, aún cuando sean monolíticos como GNU-Linux, son altamente modulares, permitiendo el desarrollo de ciertos subsistemas como *device drivers* o *filesystems* como *módulos de carga y descarga dinámica* (ver **LInux Kernel Modules**), permitiendo al administrador del sistema cargar o descargar componentes *en caliente*, es decir sin necesidad de reiniciar (*reboot*) el sistema.

> **Un principio de diseño sugerido para los SO es definir e implementar *mecanismos* en las capas mas bajas y que las *políticas* se definan en los niveles mas altos. Esto garantiza una buena *adaptabilidad* de un kernel a diferentes escenarios.**

Interfaz de un SO

Un SO presenta a los desarrolladores de aplicaciones interfaces, tanto a nivel de código fuente como en formatos binarios.

> **Application Programming Interface (API): Interfaz a nivel de *código fuente* que incluye las llamadas al sistema y las funciones (y tipos de datos) de la biblioteca estándar.**

El estándar **portable OS interface (POSIX)** define una *API* de portabilidad a nivel de sistema y usuario como también con un conjunto de *utilidades* (como *shells* y otros comandos) con el fin de compatibilizar variantes del SO *UNIX*.

En particular, éste estándar define la *API* como declaraciones de funciones y tipos de datos en `C`.

> **Application Binary Interface (ABI): Interfaz de bajo nivel que especifica los formatos binarios y convenciones para permitir la interacción de programas de usuario con las *bibliotecas* y con el *kernel* del sistema operativo.**

Los sistemas operativos generalmente especifican una *ABI* para cada plataforma de hardware (x86, x86-64, ARM, risc-v, etc). La *ABI* incluye:

1. Convenciones para las *invocaciones a funciones*.
2. Convenciones para los *syscalls*.
3. Formato de archivos binarios (*objetos* y *ejecutables*)

A modo de ejemplo, GNU-Linux usa binarios en formato **Executable and Linking Format (ELF)** para todas las plataformas soportadas. La convención para las llamadas a funciones en x86 (32 bits) es que los parámetros se *apilan* por el invocante en orden reverso (de derecha a izquierda) y en x86-64 se pasan hasta 6 argumentos en los registros `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9` (el resto en la pila en orden reverso). Los valores retornados quedan en el registro `eax/rax`, respectivamente. En ambas arquitecturas se usa las instrucciones `call/system` para una invocación.

En arquitecturas RISC, como RISC-V, los parámetros se pasan en los registros `a0, ..., a7` y el valor retornado queda en `a0`.

Estas convenciones deben ser respetadas por cualquier compilador de la plataforma.

Usuarios, sesiones, shell y variables de ambiente

Generalmente el uso de un sistema multi-usuario se basa en *sesiones*. Al comienzo un usuario debe presentar sus *credenciales* como su identificador y una *contraseña*. Luego el sistema lanza un *shell*, el cual puede basarse en una interfaz de texto (*consola*) o gráfica. Luego el usuario puede *interactuar* con el sistema lanzando otros programas para finalmente cerrar la sesión.

El *shell* define un conjunto de *variables de ambiente* que el usuario y los programas pueden usar y modificar.

Algunas variables de ambiente comunes en sistemas tipo UNIX son:

- `HOME` : Directorio o espacio de trabajo del usuario.
- `PATH` : Lista de directorios de búsqueda de programas.
- `PWD` : Directorio corriente (*print working directory*).
- `SHELL` : Programa (path) shell.
- `LANGUAGE` : Lenguaje de la interfaz de usuario

Se puede usar el comando `echo` para acceder al valor de una variable de ambiente. El valor de una variable `V` se accede con la expresión `$V` en UNIX (`%V` en MS-Windows).

Es posible listar las variables de ambiente con el comando `printenv` en UNIX o `set` en MS-Windows. Una variable de ambiente se puede crear o modificar su valor mediante el comando `var=value` o `export` (variable global), dependiendo del *shell*. En MS-Windows se puede crear/modificar una variable con el comando `set` .

El siguiente ejemplo muestra parte de la *API* que nos ofrece un SO *POSIX*.

c

```c
#include <sys/types.h>  /* system data types */
#include <unistd.h>     /* UNIX standard API */
#include <stdio.h>      /* I/O standard API  */

int main(int argc, char *argv[], char *envv[])
{
    int i;
    printf("Process id: %d\n", getpid()); /* getpid() is a syscall */
    printf("Command line arguments:\n");
    for (i = 0; i < argc; i++)
        printf("%s\n", argv[i]);
    printf("Session environment variables:\n");
    for (i = 0; envv[i]; i++)
        printf("%s\n", envv[i]);
    exit(0); /* exit syscall: process exit code */
}
```

En este ejemplo se puede observar que:

1. Un programa de usuario comienza por una función denominada `main` que recibe como parámetros el número de los argumentos de la línea de comandos, un arreglo de punteros (strings) a ellos y un arreglo (null-terminated) de strings con las variables de ambiente de la forma `variable=valor` .

2. Los programas interactúan con los dispositivos de entrada/salida (en este caso la *consola* o *terminal*) indirectamente por medio de funciones de biblioteca, las cuales a su vez ejecutan *syscalls*. En este caso `printf()`, internamente ejecuta el *syscall* `write(1, buffer, count)`. El *descriptor de archivo* cero (0) está conectado a la entrada estándar (generalmente el teclado). Los descriptores 1 y 2 refieren a la *salida* y *salida de errores* estándar, asociados a la pantalla.

3. Las llamadas al sistema (como `getpid()` o `exit(n)`) están *implementadas* como funciones de la *biblioteca estándard*.

4. Al compilar este programa (con `gcc -o myprog myprog.c`) el *linker* lo *enlaza* con la biblioteca estándar.

APIs de Sistemas Operativos

Un sistema operativo ofrece a las aplicaciones servicios que permiten acceder a recursos del sistema y facilidades como mecanismos de comunicación y sincronización entre procesos y otros.

El sistema operativo UNIX, creado por Ken Thompson y Dennis Ritchie provee interfaces simples y coherentes. La mayoría de los SO modernos se basan en estas APIs.

La API de UNIX se describe como un conjunto de *funciones C* que implementan *syscalls*. A continuación se muestra una lista parcial de llamadas al sistema correspondientes a operaciones sobre procesos y archivos.

Procesos

- `fork()` : Crea un nuevo proceso (hijo) el cual es una copia del proceso invocante.
- `exit(exit_code)` : Finaliza el proceso con código de salida `exit_code` .
- `pid wait(&exit_code)` : Espera (se bloquea) hasta que finalice un proceso hijo. Retorna el *pid* del hijo y en `exit_code` se recibe el código de salida.
- `getpid()` : Retorna el identificador del proceso corriente.
- `getppid()` : Retorna el identificador del proceso padre.
- `kill(pid)` : Envía una señal (terminación) al proceso con id `pid` .
- `sleep(n)` : Espera (se bloquea) por `n` segundos.
- `exec(filename, args)` : Carga el programa `filename` y lo ejecuta. Reemplaza la imagen de código y datos de la memoria del proceso corriente.
- `sbrk(n)` : Asigna `n` bytes de memoria adicionales al proceso corriente.

Archivos

- `open(filename, mode)` : Abre un archivo en el modo (*read*, *write*, ...) dado.
- `read(fd, buffer, count)` : Lee `count` bytes del archivo en `buffer` .
- `write(fd, buffer, count)` : Escribe `count` bytes de `buffer` al archivo.
- `close(fd)` : Cierra el archivo (libera recurso).
- `dup(fd)` : Dumplica (copia) `fd` en el primer descriptor no usado.
- `pipe()` : Crea un *pipe* para comunicación entre procesos.
- `chdir(dirname)` : Cambia de directorio.
- `mkdir(dirname)` : Crea un directorio.
- `mknod(name, major, minor)` : Crea un archivo *especial* (device).
- `fstat(fd)` : Retorna el *estado* y metadatos (size, ...) del archivo.
- `link(f1, f2)` : Crea un *alias* (`f2`) para el archivo `f1` .
- `unlink(filename)` : Borra el archivo.

En algunas versiones modernas de SO tipo UNIX algunas de estas *llamadas al sistema* se han extendido ampliamente, como por ejemplo en GNU-Linux.

Todas las llamadas al sistema se presentan al programador como funciones de biblioteca. Su implementación dispara un `syscall` , es decir una entrada a un punto del kernel. A partir de allí se ejecuta código del kernel hasta su retorno.

Todas las llamadas al sistema retornan un entero. Generalmente en caso de falla retornan un valor negativo (comúnmente -1).

Interfaz del usuario

Un SO presenta algún tipo de interfaz al usuario. Generalmente el programa que se encarga de interactuar con el usuario es un *shell*, el cual puede ser basado en *línea de comandos* (como *bash*, *zsh*, *csh* de UNIX o *cmd* en MS-Windows).

Un *shell* de línea de comandos permite al usuario ingresar comandos para ejecutar programas y soportan operadores sobre el control, sincronización y comunicación entre los procesos creados.

El shell se dispara al inicio del sistema, generalmente luego que el usuario inicia su sesión (*login*) asociada a la *consola* o *terminal*.

Típicamente un *shell* presenta al usuario un *prompt* que puede incluir su identificador de usuario, finalizado comúnmente por el caracter `#`. Cabe aclarar que el prompt de un shell es configurable por el usuario (Ej: `.bash_profile` en sistemas tipo UNIX).

El sistema presenta al usuario un *sistema de archivos*. Hay archivos de diferentes tipos:

1. *Programas*: archivos ejecutables (o *comandos*).
2. *Datos*: Contienen datos en algún formato (binario o texto).

Un archivo de textos contiene una secuencia de códigos de caracteres (ej: ASCII, UTF-8, ...). Generalmente se estructuran en líneas, es decir, secuencias de caracteres finalizados por `\n` (*newline*) o `\n\r` (*newline*/*carriage-return*).

El conjunto de archivos del sistema se organizan en una estructura jerárquica (de árbol) con *directorios* (o *carpetas*), los cuales a su vez contienen otro subárbol de directorios y archivos.

Para hacer referencia a un archivo se puede hacer de dos formas:

1. *Full path*: Camino desde la raíz del sistema de archivos hasta el archivo.

   ejemplo: `cat /home/user/docs/myfile.txt`

2. *Relativa*: En base al *directorio corriente*, es decir, el directorio de trabajo actual al que se accede mediante el comanddo `cd dir-path`).

   Ejemplo: `cd docs ; cat myfile.txt`

En los sistemas tipo UNIX la convención es que un proceso con *código de salida=0* significa *terminación normal*, mientras que otros valores representan algún código de error, dependiendo de la aplicación.

Un shell típico en sistemas tipo UNIX reconoce los siguientes operandos y operadores:

- Operandos: Pueden ser un *comando* o un *nombre de archivo*
- Operadores:
  - `cmd1 ; cmd2` : Ejecuta `cmd1`, al finalizar ejecuta `cmd2`.
  - `cmd1 && cmd2` : Ejecuta `cmd2` sólo si `cmd1` finaliza exitosamente.
  - `cmd1 || cmd2` : Ejecuta `cmd2` sólo si `cmd1` finaliza con error.
  - `cmd &` : Lanza el `cmd` de fondo (en *background*), el shell no espera a su terminación, devolviendo el *prompt* inmediatamente.
  - `cmd n> file` : Se *redirige* el *descriptor de archivo de salida* `n` al archivo `file`. Si `n` se omite, se asume 1 (salida estándar).

- `cmd < file` : Redirige la entrada estándar (comúnmente el teclado) desde el archivo `file` .
- `cmd1 | cmd2` : Ejecuta `cmd1` y `cmd2` concurrentemente, *redirige* la salida estándar de `cmd1` a un *pipe* y la entrada estándar de `cmd2` a la *salida del pipe*. Más abajo se dan mas detalles sobre *pipes*.

Procesos

En esta sección se analizan las llamadas al sistema para la creación, destrucción,sincronización y comunicaciones entre procesos (*Inter Process Communication - IPC*).

En una API tipo UNIX la única llamada al sistema para crear un nuevo proceso es `fork()` , la cual crea una copia del proceso corriente, conjuntamente con su estado.

El estado de un proceso (mantenido por el kernel) tiene generalmente los siguientes atributos:

- Su estado: `RUNNING` , `READY` , `SLEEPING` , `TERMINATED` ...
- Su *identificador* o *pid*: Un número > 0
- Su *imagen de mamoria*: Código, datos globales y stack.
- El conjunto (tabla) de *archivos/pipes* abiertos.
- Una referencia a su padre (*parent*)
- Un stack para ejecución en modo *kernel*
- Espacio para *salvar* su *contexto* (estado de la CPU) cuando cambie de estado.
- El código de salida o terminación.

La llamada al sistema `fork()` , ejecutada inicialmente por un proceso *padre* hace que el kernel cree una copia (el *hijo*) del proceso corriente, el cual *hereda* su estado, en particular la *tabla de archivos abiertos*.

Al retornar de la llamada al sistema, en el proceso padre retorna el `pid` (> 0) del nuevo proceso (*hijo*).

El nuevo proceso (hijo), al heredar el estado del padre, inicia su ejecución en el retorno del `fork()` , aunque aquí retorna 0.

El siguiente ejemplo muestra el uso de `fork()` .

c

```c
#include <unistd.h> /* UNIX syscalls */
#include <stdio.h>  /* portable i/o functions */

int main(void) {
    int pid = fork();
    if (pid == 0) {
        printf("I'm the child. Pid: %d\n", getpid());
    } else {
        printf("I'm the parent. Pid: %d\n", getpid());
        printf("My child is: %d\n", pid);
    }
}
```

Un proceso *padre* debería esperar hasta que sus hijos terminen antes de finalizar.

A continuación se muestra un ejemplo del uso de `wait` y `exit` .

```c
#include <unistd.h> /* UNIX API (syscalls) */
#include <stdio.h>  /* portable i/o functions */

int main(void) {
    if (fork() == 0) {
        printf("I'm the child. Pid: %d\n", getpid());
        exit(0);
    } else {
        int child, status;
        printf("I'm the parent. Pid: %d\n", getpid());
        child = wait(&status);
        printf("In parent: child %d finished\n", child);
    }
}
```

La llamada al sistema `exec(program, args)` reemplaza la imagen de memoria (código y datos) del proceso corriente con el código y datos de `program`. Cabe aclarar que `exec()` no crea un nuevo proceso. El proceso comienza su ejecución desde el `entry point` (*function address*) que está especificado en el archivo ejecutable `program`.

Es común que un programa *lance* otro programa para realizar alguna tarea para luego continuar otras operaciones. En este modelo se debe hacer siguiendo el patrón mostrado en el siguiente ejemplo, el cual ejecuta el comando `ls -l`:

```c
#include <unistd.h> /* UNIX syscalls */
#include <stdlib.h> /* wait() */
#include <stdio.h>  /* portable i/o functions */

int main(void) {
    char *args[] = {"ls", "-l", 0};

    if (fork() == 0) {
        /* in child */
        execve("/bin/ls", args, 0);
        /* on success, below is unreachable code */
        printf("Ooops, exec() failed!\n");
        exit(-1);
    } else {
        wait(0);
        printf("In parent: child finished\n");
    }
}
```

Shell: Control de procesos

Un comando de la forma `cmd &` lanza el programa `cmd` como un *proceso de fondo (background)*. El shell le asigna un *identificador de job* como se muestra abajo.

sh

```
$ ping google.com > output &
[1] 4287
$ _
```

El shell muestra el *job id* (ejecutando en background) 1 asociado al proceso con pid 4287.

El comando `jobs` listará los procesos en background y su estado.

sh

```
[1]+  Running                 ping -i 5 google.com &
```

Un proceso en *foreground* (el shell está esperando a que termine) puede *suspenderse* o pararse usando `Ctrl-Z` . El shell devuelve el *prompt* y puede usarse el comando `bg` para continuarlo de fondo.

Es posible *finalizar* (*matar*) un proceso de fondo mediante el comando `kill %job_id` .

sh

```
$ ping google.com
PING google.com (74.125.226.71) 56(84) bytes of data.
64 bytes from lga15s44-in-f7.1e100.net (74.125.226.71): icmp_seq=1 ttl=55 time=12.3 ms
64 bytes from lga15s44-in-f7.1e100.net (74.125.226.71): icmp_seq=2 ttl=55 time=11.1 ms
64 bytes from lga15s44-in-f7.1e100.net (74.125.226.71): icmp_seq=3 ttl=55 time=9.98 ms
Ctl-Z
[1]+  Stopped                 ping google.com
$ bg
[1] 4579
...
kill %1
[1]+  Finished                ping google.com
```

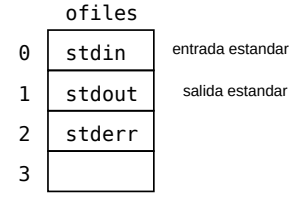
Redirección de entrada/salida

Cada proceso inicia comúnmente con tres archivos abiertos (heredados) desde `init` , el primer proceso del sistema, lanzado por el *kernel* luego del *boot*.

El *file descriptor 0* (*standard input*) está asociado a la entrada de la *consola*, comúnmente un *teclado*.

Los descriptores 1 y 2 (*standard output and error*, respectivamente) están asociados a la salida de la *consola*, típicamente la pantalla o *display*.

Es posible ver los *archivos abiertos* de un proceso como una tabla de la forma



La llamada al sistema `open(filename, mode)` retorna un nuevo descriptor de archivo, el cual puede verse como un *índice* de la tabla *ofiles* (*opened files*).

índice de la tablita

Este modelo de dos pasos `fork()/exec()` permite al proceso padre *modificar el ambiente de ejecución* del nuevo programa a lanzar. En particular puede *redireccionar* las entradas y salidas.

EL fork hace que el nuevo proceso herede los archivos abiertos del padre.

El siguiente ejemplo permite muestra cómo es posible lanzar un proceso hijo que en lugar de escribir en la entrada estándar lo hará en el archivo `/tmp/out.txt` .

c

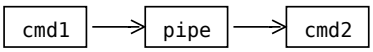
```c
#include <unistd.h> /* UNIX syscalls */
#include <fcntl.h>  /* file operations and flags */

int main(void) {
    if (fork() == 0) {
        /* in child: open/create file with permissions (user: rw, group:r) */
        int fd = open("out.txt", O_WRONLY | O_CREAT, 0640);
        printf("File fd: %d\n", fd);    /* fd should have the value 3 */
        close(1);                       /* close stdout */
        dup(fd);                        /* duplicate fd: ofiles[1] == fd */
        write(1, "Hello world\n", 12);
    }
}
```

Pipes

Un *pipe* es un mecanismo de comunicación (unidireccional) entre procesos. Un comando de shell de la forma `cmd1 | cmd2` hace que se lancen concurrentemente `cmd1` y `cmd2` previa redirección de la salida estándar al extremo de escritura del *pipe* y la entrada estándar de `cmd2` al extremo de lectura del pipe. Gráficamente:

Es como productor-consumidor que vimos en Comparativo



La llamada al sistema `pipe(p)` retorna en el argumento de salida `p` dos *file descriptors* (enteros). El primer elemento del arreglo corresponde al extremo de lectura del pipe y el segundo al de escritura. El siguiente ejemplo, muestra su uso.

c

```c
#include <unistd.h> /* UNIX syscalls */
#include <stdio.h>

#define N 20

int main(void) {
    int p[2];
    pipe(p);    /* create pipe */

    if (fork() == 0) {
        /* in child: read from pipe */
        char buffer[N];
        close(p[1]);            /* close write pipe descriptor */
        read(p[0], buffer, N);
```

El hijo lee en el pipe y el padre escribe en el pipe.
Es como mandar un mensaje.

Cerramos el pipe de escritura ya que el hijo solo lee.

```
            printf("Child read: %s", buffer);
        } else {
            /* in parent */
            close(p[0]);            /* close read pipe descriptor */
            write(p[1], "Hello from parent\n", 18);
        }
    }
```

Cerramos el pipe de lecutra ya que el padre solo escribe

Aca deberia ser 19, ya que falta el caracter /0 que siempre tiene c

Los pipes son unidimensionales, si queremosq ue el hijo le responda al padre deberiamos crear otro.,

Si me quiero comunica un proceso con otro que no es un hijo, se puede utilizar un FIFO. (ES como crear un archivo y se utiliza la se;al MKFIFO crea un fifo, este archivo no contiene datos, solo se utiliza como un "PIPE")
Ej: echo "hello world" > mypipe
Aca el pipe se bloquea.
Em ptra ter,omal podemos leeer cat mypipe
Asi permitiendo que el otro termine.

Señales

Las señales son un mecanismo básico de *comunicación entre procesos* para implementar la notificación de *eventos*. El kernel en algunos casos abstrae una *interrupción* o *excepción* en una *señal* enviada a un proceso. También un proceso puede enviar una *señal* a otro proceso. Este mecanismo fue desarrollado en las primeras versiones de UNIX en la década de 1970.

La llamada al sistema `kill(pid, signal)` envía la señal `signal` al proceso identificado con `pid` . Cada señal tiene ide que sea desde ntificadores numéricos. El sistema envía señales a los procesos cuando el usuario en una terminal pulsa las combinaciones de teclas como por ejemplo `Ctrl-C` , la cual envía `SIGINT` (*interrupt*) o `Ctrl-Z` , que envía `SIGTSTP` .

Cuando el proceso no esta hecho para recibir una se;al basicamente mata el proceso que deberia tomarlo. ES decir el proceso deberia tener la capacidad de obtener una se;al.

Un proceso puede *manejar* una señal *s* instalando un *signal handler* por medio del syscall `signal(sig, handler)` , donde `sig` es la señal a manejar y `handler` es una función de la forma `void handler(int signal)` .

Por omisión cada señal tiene un *default handler* implementado en el kernel. Su comportamiento depende de cada señal. Por ejemplo, para la señal `SIGINT` el *default handler* finaliza el proceso, mientras que el *default handler* para `SIGTSTP` causa que el proceso se *suspenda* (queda en estado `SLEEPING`).

Cuando un proceso recibe una señal, se altera su flujo de ejecución normal y se dispara su *handler*. Luego del retorno, el proceso continúa con su ejecución en el estado que estaba antes de la recepción de la señal.

La única señal que no puede ser atrapada (manejada) es `SIGKILL` la cual se usa para *terminar* un proceso (si es que el usuario es el *usuario efectivo*, es decir, quien lo ejecutó).

Alarmas

Es como una especie de timeout, da el tiempo para tirar la se;al

se programa una alarma.

Las llamadas al sistema `alarm(seconds)` y `setitimer(...)` permiten definir una alarma, es decir que luego de transcurrido ese tiempo, el kernel lanzará la señal `SIGALRM` al proceso.

Una alarma generalmente se usa cuando una aplicación realiza una operación (por ejemplo, enviar un mensaje) y si no tiene respuesta dentro de un intervalo de tiempo (*timeout*), requiere realizar alguna acción de recuperación o re-intento.

Trazado de procesos

Un proceso puede controlar la ejecución de otro proceso. Al primero se lo conoce como el *trazador* y al otro el *trazado*. Comúnmente un *debugger* se implementa usando este mecanismo, entre otros.

La llamada al sistema `ptrace(request, pid, addr, data)`. El proceso *trazado* se suspende cada vez que envíe una señal o haga una llamada al sistema. El *trazador* será notificado en un `waitpid(pid, &status)`. En `status` se indica la causa de la suspensión del proceso trazado.

Luego, el *trazador* puede realizar alguna de las siguientes acciones sobre el *trazado*:

- Identificar qué llamada al sistema realizó.
- Leer o escribir en el área de código o datos. Así un debugger, puede por ejemplo, definir un *breakpoint* por software, reemplazando una instrucción por una llamada al sistema.
- Acceder al estado de la CPU (valores de los registros) del proceso bloqueado
- Hacer que continúe la ejecución
- Otras. Para más detalles, hacer `man ptrace`.

Anterior
Introducción

Próximo
Herramientas de desarrollo