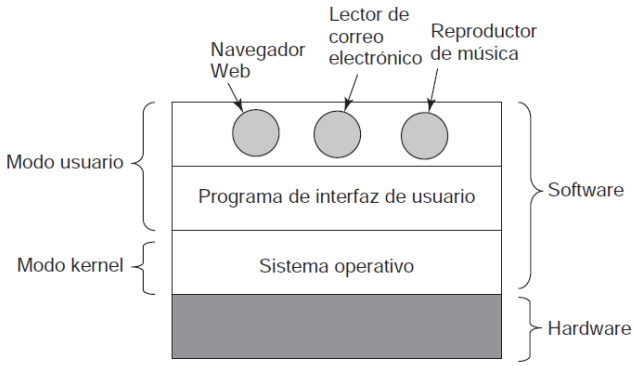
SISTEMAS OPERATIVOS

UNIDAD 1: Introducción

Las computadoras están formadas por un sistema operativo cuya finalidad es proporcionar a los programas de usuario un modelo de computadora mejor y más simple, encargándose también de la administración de recursos como lo son los procesadores, memorias, discos, teclados, mouse, etc. El programa con el que los usuarios interactúan se llama *shell* cuando está basado en texto y *GUI* para interfaces gráficas. La mayoría de las computadoras tienen dos modos de operación: *modo kernel* y *modo usuario*. El sistema operativo es la pieza fundamental del software y se ejecuta en modo kernel (modo supervisor). En este modo el sistema operativo tiene acceso completo a todo el hardware y puede ejecutar cualquier instrucción que la máquina sea capaz de ejecutar. El resto del software se ejecuta en modo usuario, donde solo se permite ejecutar un subconjunto de las instrucciones de máquina.

¿Qué es un sistema operativo?
Básicamente es el software que se ejecuta en modo kernel, aunque no siempre es así.
El SO tiene dos roles básicos que no están relacionados: proporcionar a los programadores de aplicaciones recursos simples en vez de complejos conjuntos de hardware, y administrar esos recursos de hardware. En otras palabras *abstracción* y *gestión de recursos*. Entonces un sistema operativo es el software que gestiona los recursos de la computadora para los usuarios y sus aplicaciones.



(En la figura, las de arriba básicamente son las aplicaciones).

La *abstracción* se refiere a que el usuario no desea involucrarse con la programación de los discos duros, en lugar de ello lo que se hace es una abstracción simple de alto nivel que se encargue de lidiar con el disco. El trabajo del SO es crear buenas abstracciones para después implementar y administrar los objetos abstractos creados. Una de las principales tareas del SO es ocultar el hardware y presentar a los programas abstracciones agradables con las que se pueden trabajar, básicamente ocultan la parte fea con la parte hermosa.
En los sistemas operativos de uso general los usuarios interactúan con las aplicaciones, las aplicaciones se ejecutan en un entorno proporcionado por el sistema operativo y el sistema operativo media para el acceso al hardware. El nivel más bajo (*hardware*) proporciona procesadores, memoria y todos los dispositivos de comunicación con el exterior como teclado, impresora, monitor, etc. Luego el sistema operativo se ejecuta como la capa de *software* más baja. Contiene una capa que gestiona los dispositivos de hardware y servicios relacionados a las

aplicaciones. El sistema operativo funciona en un entorno de ejecución separado al de las aplicaciones.

Al SO se lo relaciona con 3 funciones básicas:

- *Referee (árbitro)*: porque gestiona los recursos compartidos entre las diferentes aplicaciones que se ejecutan en una misma máquina física, pudiendo por ejemplo, detener un programa y empezar otro. El SO aísla las aplicaciones entre sí. Como las aplicaciones comparten los recursos físicos, el SO decide qué aplicaciones y cuándo consiguen los recursos. El SO debe permitir que múltiples aplicaciones se ejecuten al mismo tiempo.
- *Ilusionist (ilusionista)*: porque realiza una abstracción de hardware físico para simplificar el diseño de aplicaciones. Los SO provocan la ilusión de la memoria casi infinita, a pesar de tener una cantidad limitada de memoria física. Estas ilusiones son las que permiten escribir aplicaciones independientemente de la cantidad de memoria física en el sistema o el número de procesadores.
- *Glue (pegamento)*: proporciona un conjunto de servicios comunes que facilitan el intercambio de información entre aplicaciones. Cortar y pegar funcionan de manera uniforme en todo el sistema, de modo que un archivo escrito por una aplicación puede ser leído por otra.

Ampliación del rol de referee

Un ejemplo para diferenciar la ejecución en la máquina física y la máquina virtual o abstracta que es la que proporciona el SO, es qué debería ocurrir si una aplicación se ejecuta en un bucle infinito [*while (true)*]. Si los programas corrieran en el hardware, este bucle podría bloquear al equipo sin darle entrada al usuario. Si el SO se asegura que cada programa tenga su propia porción de recursos, una aplicación específica se podría bloquear, pero dejando trabajar a las demás simultáneas, además de que el usuario podría pedir al SO forzar al programa a salir del bucle.

Un error en una aplicación no debe perturbar a las demás o al propio SO, esto se conoce como el *aislamiento de fallos (isolation)*. Por eso recibe el nombre de *árbitro*, ya que separa los conflictos y facilita el intercambio entre las diferentes aplicaciones.

Ampliación del rol de ilusionista

La *virtualización* proporciona una aplicación con la ilusión de recursos que no están físicamente presentes. Por ejemplo, el SO puede proporcionar la abstracción de que cada aplicación tiene un procesador propio, a pesar de que a nivel físico solo haya un único procesador que se comparte entre todas las aplicaciones. La mayoría de los recursos físicos pueden ser virtualizados, como el ejemplo dado más arriba de que la memoria física es finita y se da la ilusión de una memoria infinita para las aplicaciones.

Algunos SO virtualizan todo el equipo y ejecutan un sistema operativo como una aplicación, sobre otro sistema operativo. Esto es lo que se conoce como *máquina virtual*. El SO que se ejecuta en la máquina virtual se llama *SO huésped*. Se simula que se está ejecutando en una máquina real y física, pero esto es solo una ilusión creada por el verdadero SO.

Los *desafíos* que tiene el SO básicamente son:

- *Confiabilidad*: que realice lo que tenga que hacer.
- *Disponibilidad*: tiempo medio entre fallas y recuperación.
- *Seguridad*: acceso indebido del SO y aplicaciones.
- *Privacidad*: acceso a los datos por usuarios autorizados.
- *Portabilidad*: aplicaciones y hardware.
- *Tiempo de respuesta*.
- *Rendimiento*: cuantas operaciones pueden realizarse simultáneamente.
- *Equidad*: en la distribución de recursos.
- *Predictabilidad*: consistencia en el tiempo.

Para diferenciar los sistemas operativos y las aplicaciones tenemos en cuenta los niveles de protección: el hardware soporta varios niveles de privilegio, donde diferente software corre en diferentes niveles. El nivel de protección se traduce en instrucciones a ejecutar, gestión de direcciones y espacio de memoria accesible. Dentro de los privilegios el hardware provee de al menos dos niveles: *Usuario* (el que ejecuta las aplicaciones) y *Supervisor* o *modo protegido* (que es el sistema operativo).
Por ejemplo, el Intel x86 posee 4 niveles de privilegio, dividido en segmentos de memoria con niveles de privilegio, donde los niveles 0, 1 y 2 se ejecutan en modo supervisor, donde el 1 y 2 a su vez manejan los drivers de los dispositivos, y el nivel 3 se encarga del modo usuario. Con el mayor nivel de privilegio no es posible invocar directamente el código. El procesador a su vez utiliza estos niveles de privilegio para saber que puede hacerse con los datos y código.

Arquitectura de los Sistemas Operativos
Dentro de las arquitecturas (interior del sistema operativo) básicas tenemos, por ejemplo:

- *Ejecutivos:* el sistema operativo es un conjunto de servicios, donde las aplicaciones pueden invocar dichos servicios en forma directa; se comparte el espacio de memoria. Uno de los más usados por su facilidad de manejo y buena funcionalidad. Se ejecuta en modo kernel y excepto por la interfaz de usuario, constituye un sistema operativo en sí mismo.

- *Monolíticos:* es la organización más común, donde todo el sistema operativo se ejecuta como un solo programa en modo kernel. Cada procedimiento en el sistema tiene la libertad de llamar a cualquier otro. Dicho de otra forma, la capa N puede utilizar los servicios de la capa N-1. Para construir el programa objeto primero se compilan todos los procedimientos individuales y luego se vinculan en conjunto para formar un solo archivo ejecutable, usando el enlazador del sistema. No oculta información, todos los procedimientos son visibles para cualquier otro procedimiento. Para cada llamada al sistema hay un procedimiento de servicio que se encarga de la llamada y la ejecuta. La estructura básica es un programa principal que invoca el procedimiento del servicio solicitado, un conjunto de procedimientos de servicio que llevan a cabo las llamadas al sistema y un conjunto de procedimientos utilitarios que ayudan a los procedimientos de servicio.

- *Micro Kernels:* con el diseño de capas se puede elegir que colocar en modo kernel y cuáles no, donde en modo protegido o kernel sólo se colocan los componentes esenciales. Tradicionalmente todas las capas se incluían en kernel pero es mejor poner lo menos posible ahí, ya que los errores en el kernel pueden paralizar el sistema. La idea básica del micro kernel es lograr una alta confiabilidad al dividir el sistema operativo en módulos pequeños y bien definidos, donde sólo uno (micro kernel) se ejecuta en modo kernel y el resto se ejecuta como procesos de usuarios ordinarios. Por ejemplo, al ejecutar un driver de un dispositivo como un proceso separado, un error hace que falle ese componente sólo y no lo haga todo el sistema, a diferencia del monolítico con todos los drivers en kernel, donde un error puede hacer referencia a una dirección de memoria inválida y provocar un error en el sistema. El uso de micro kernel es muy común en aplicaciones de tiempo real, industriales y militares. Fuera del kernel el sistema se estructura como tres capas de procesos, donde todos se ejecutan en modo usuario. La capa más baja contiene los drivers de dispositivos, luego está la capa que contiene los servidores, los que realizan la mayor parte del trabajo del sistema operativo y la capa más alta que maneja los programas de usuario.

- *Máquina Virtual:* es un mecanismo que permite crear la ilusión de varios ambientes de ejecución en un único ambiente de hardware. Es como trabajar con una máquina física dentro de un sistema operativo. Es decir, es un software que simula a una computadora y puede ejecutar programas como si fuese una computadora real. Es un duplicado y aislado de una máquina física. El mismo puede ejecutarse con otro sistema operativo distinto al que se posee en la máquina física. Los procesos que se ejecutan están limitados por los recursos y las abstracciones proporcionados por ella, es decir, que no se pueden escapar de la máquina virtual.

- *Exo Kernels:* en vez de clonar la máquina actual como lo hace la máquina virtual, otra estrategia es particionarla, es decir a cada usuario se le proporciona un subconjunto de los recursos. En la capa inferior que se ejecuta en modo kernel hay un programa llamado exo kernel, donde su trabajo es asignar recursos a las máquinas virtuales y después comprobar los intentos de utilizarlos para que ninguna máquina trate de usar recursos de otra. Es un sistema operativo implementado como máquina virtual con recursos limitados. El exo kernel ahorra una capa de asignación. En los otros diseños, cada máquina virtual piensa que tiene un disco propio. donde cada máquina debe tener tablas para reasignar las direcciones del disco. En el exo kernel esto no es necesario ya que sólo lleva el registro para saber a cuál máquina virtual se le ha asignado cierto recurso.

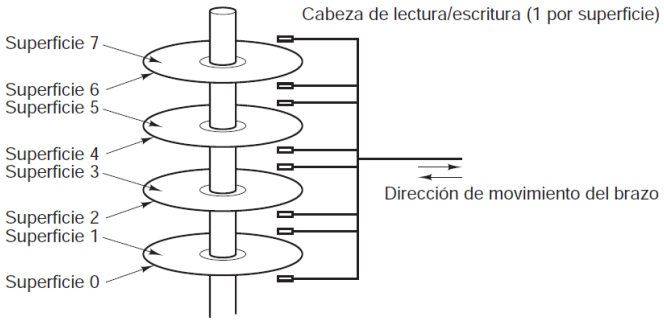
- *Sistemas Operativos Extensibles:* los sistemas cliente-servidor lo que hacen es eliminar la mayor parte del kernel posible. El método opuesto es colocar más módulos en kernel pero de forma protegida.

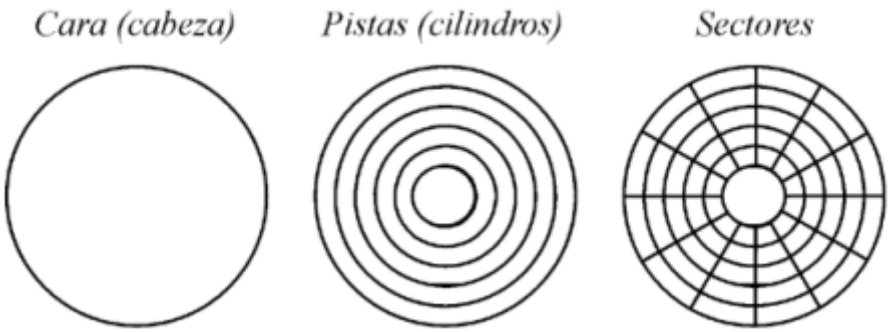
UNIDAD 2: Sistema de Archivos - File System

Antes de empezar se necesita conocer como está conformado un disco y sus características esenciales.

Disco

Contiene uno o más *platos* que giran desde 5400 hasta 10800 rpm. Contiene un *brazo mecánico (cabeza de lectura/escritura)* con un punto de giro colocado en una esquina, mientras se mueve el disco sucede algo similar a un toca disco. La información se escribe en el disco en una serie de círculos concéntricos. En cualquier posición del brazo cada una de las cabezas puede leer una región conocida como *pista* y en conjunto todas las pistas para una posición dada del brazo forman un *cilindro*. Cada pista se divide en *sectores*, por lo general de 512 bytes cada uno. Cuando se desplazan los brazos, una vez que el mismo se encuentra en la pista correcta, se debe esperar a que el sector necesario gire debajo de la cabeza. Entonces un sector tiene 512 bytes y un *clúster* (unidad de asignación) tiene un conjunto de sectores (1, 2, 4, 8, 16, …, es decir 2^n).





El *sector 0* es el primer sector de la primera pista sobre el cilindro más externo. El mapeo procede en orden a través de esa pista, luego el resto de las pistas en el cilindro y luego el resto de los cilindros desde el más externo al más interno.

Espacio de direcciones

Cada computadora tiene una memoria principal para mantener los programas en ejecución. En un SO simple, solo hay un programa en memoria a la vez. Para ejecutar un segundo programa se debe quitar el primero y colocar el segundo en memoria. Los SO más sofisticados permiten colocar varios programas a la vez en memoria. Hoy en día existe la memoria virtual, en la cual el SO mantiene una parte del espacio de direcciones en la memoria principal y otra parte en el disco, moviendo pedazos de un lugar a otro según sea necesario. La memoria virtual sirve para ejecutar programas más extensos que la memoria física de la computadora, llevando y trayendo pedazos entre la RAM y el disco.

Concepto de archivo (file system)
Es una colección de información relacionada y almacenada en un dispositivo de almacenamiento secundario con espacio de direcciones lógicas contiguas.

La *estructura interna (lógica)* puede ser:

- Una secuencia de bytes donde el sistema no sabe ni le importa que haya en el archivo, sólo ve bytes. Tanto Unix como Windows usan esta metodología.
- Una secuencia de registros de longitud fija, donde la operación de lectura devuelve un registro y la de escritura sobrescribe o agrega un registro.
- Una secuencia de registros de longitud variable.

Dentro de los *tipos de archivos* podemos encontrar:

- Regulares: contienen información del usuario.
- Directorios: sistemas de archivos que mantienen la estructura del sistema de archivos.
- De dispositivo: modelan dispositivos de E/S en serie como impresoras y redes, por ejemplo.

La *forma de acceso* a los archivos puede ser:

- Secuencial: un proceso puede leer todos los bytes o registros en un orden, empezando desde el principio y sin poder saltear pero si rebobinar. Era mejor cuando se usaban cintas magnéticas en lugar de discos.
- Aleatorio: leer bytes o registros fuera de orden, especialmente en los discos.

En cuanto a los *atributos*, todos los archivos poseen nombre y datos. El nombre es la única información en formato legible, luego tenemos: el tipo (cuando el sistema soporta diferentes tipos), la localización que contiene un puntero de donde está ubicado el archivo en el dispositivo, el tamaño actual del archivo, una protección para determinar quién puede leer, escribir y ejecutar el mismo, y un tiempo y fecha de identificación del usuario necesario para la protección, seguridad y monitoreo.

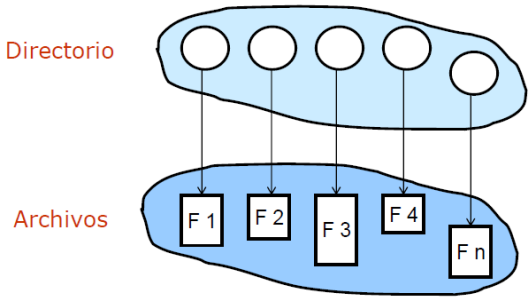
Dentro de las *operaciones* que podemos realizar con los archivos podemos incluir las de gestión (crear, borrar, renombrar, copiar, establecer y obtener atributos) y las de procesamiento (abrir y cerrar, leer, escribir, ya sea insertando o modificando información).

Se necesitan varios datos para la administración de los *archivos abiertos*: un puntero del archivo (puntero a la última locación read/write), un contador de archivo abierto teniendo en cuenta el número de veces que el archivo es abierto, la localización del archivo en el disco, y la información de los derechos de acceso.

Los archivos pueden tener distintas *extensiones*, como por ejemplo: ejecutables (.exe .com .bin) que son programas en lenguaje de máquina listos para correr, de texto (.txt .doc) que son documentos con datos textuales, archivos (.zip .rar) agrupados en un solo archivo, etc.

Estructura de directorios
Para llevar el registro de los archivos, los sistemas de archivos por lo general tienen directorios o más conocidos como carpetas, que en muchos sistemas son también archivos. Tanto la estructura de directorios como los archivos residen en el almacenamiento secundario. La organización lógica de directorios debe proporcionar eficiencia para localizar rápidamente un archivo, denominación que es adecuada para los usuarios, ya que dos usuarios pueden tener el mismo nombre para diferentes archivos y el mismo archivo puede tener varios nombres, y agrupación de los archivos

lógicamente, por ejemplo, agrupar todos los programas en C. Los sistemas de directorio pueden ser de un solo nivel, o sea un solo directorio que contiene a todos los archivos, también llamado *directorio raíz*. De esta forma se encuentran problemas de denominación y agrupación. Luego tenemos los directorios de dos niveles, con nombres de caminos o rutas, donde diferentes usuarios pueden tener archivos de igual nombre, no hay posibilidad de agrupación. También los directorios pueden estar estructurados en árboles, donde se necesita una búsqueda eficiente, con posibilidad de agrupación, con nombres de caminos *absolutos* (ruta desde el directorio raíz al archivo) y *relativos* (un usuario puede designar un directorio como el directorio de trabajo actual, en cuyo caso todos los nombres de las rutas que no empiecen en el directorio raíz se toman en forma relativa al directorio de trabajo). Finalmente puede estructurarse en grafos, donde se produce una compartición de subdirectorios y archivos, siendo más flexible y complejo.



Protección de archivos

Sirve para proporcionar un acceso controlado a los archivos, determinando lo que puede hacerse y por quién. Los tipos de acceso son para leer, escribir, ejecutar, añadir, borrar y listar. La principal solución a la protección es hacer el acceso dependiente del identificativo del usuario. Las listas de acceso de usuarios individuales tienen el problema de la longitud. Una propuesta alternativa es asociar una contraseña al archivo, el problema es recordar todas y si solo se establece una brindaría acceso total o ninguno.

Semánticas de consistencia

Es responsabilidad del SO administrar la seguridad del sistema de manera que el archivo sea accesible solo por usuarios autorizados. Especifica cuando las modificaciones de datos por un usuario se observan por otros usuarios. Por ejemplo, en Unix la escritura de un archivo es directamente observable y existe un modo en que los usuarios comparten un puntero actual del posicionamiento en un archivo. Luego tenemos los archivos *inmutables*, que cuando un archivo se declara compartido no se puede modificar.

Funciones básicas del sistema de archivos

Tener conocimiento de todos los archivos del sistema, controlar la compartición y forzar la protección de los archivos, gestionar el espacio del sistema de archivos (asignación y liberación del espacio de un disco), y traducir las direcciones lógicas del archivo a direcciones físicas del disco.

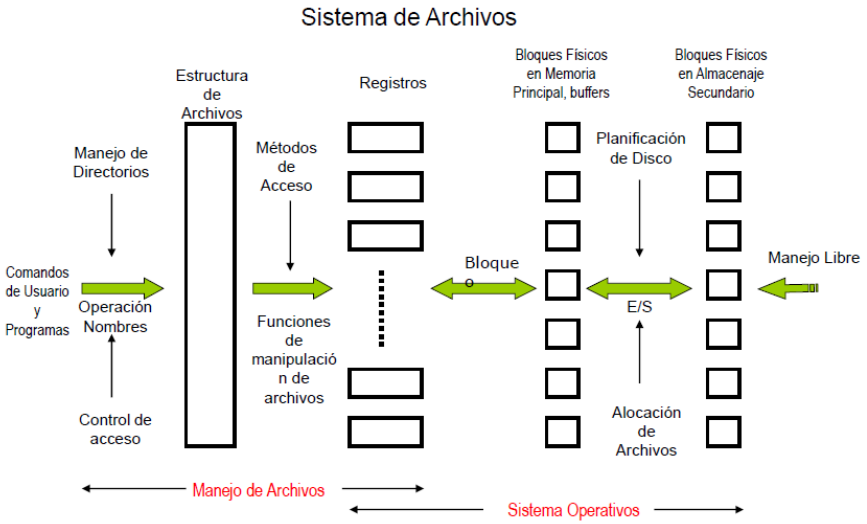
Implementación del sistema de archivos

Aquí nos encargamos de la forma en que se almacenan los archivos y directorios, como se administra el espacio en el disco y cómo hacer que todo funcione con eficiencia y confiabilidad. Los SO tienen su propio sistema de archivos y pueden ser representados en *forma textual* (por ejemplo: Shell de DOS) o en *forma gráfica* (por ejemplo: explorador de archivos de Windows).

Lo que hay que tener en cuenta para la implementación de un sistema de archivos es mantener un registro de que bloques del disco van con cual archivo. Un sistema de archivos posee dos problemas de diseño diferentes:

- Definir cómo debe ver el usuario el sistema de archivos (definir un archivo y sus atributos, definir las operaciones permitidas de un archivo, definir la estructura de directorio).
- Definir los algoritmos y estructuras de datos que deben crearse para establecer la correspondencia entre el sistema de archivos lógicos y los dispositivos físicos donde se almacenan.

Para la estructura del sistema de archivos se puede implementar, como ya se vio más arriba para los sistemas operativos, la organización en niveles o capas. Por eficiencia el SO mantiene una tabla indexada de los archivos abiertos (*descriptor de archivos*).



Existen varios *tipos de sistemas de archivos*, como por ejemplo:

- FAT: admitido por casi todos los SO. Este sistema no acepta discos duros de más de 2GB y los clúster son enormes. Existen la FAT12, FAT16 y FAT32. (Ampliado más abajo).
- FAT32: es una evolución del FAT, donde los discos pueden llegar hasta 2TB y los clúster son más pequeños.
- NTFS: actualmente reemplaza al FAT. No es factible en discos menores a 400MB ya que requiere mucho espacio para la estructura del sistema. El tamaño del clúster no depende del tamaño del disco, entonces puede elegirse libremente. Este sistema repara automáticamente los sectores dañados. Es un sistema ideal para particiones de gran tamaño. Tiene como desventajas utilizar gran cantidad de espacio en disco para sí mismo, no es compatible con DOS y SO Windows viejos, no se puede usar en disquetes, entre otras. En cuanto a sus ventajas con respecto a la FAT es que usa estructura de datos avanzadas (árboles), optimizando el rendimiento y aprovechando el espacio en disco. También mejora la seguridad.
- Linux EXT2: no es reconocido por ningún sistema Windows. Discos hasta 2GB. Su principal desventaja es que no posee registro por diario (se almacena la información necesaria para poder restablecer datos afectados por la transacción en caso que esta falle). Posee una tabla similar al FAT de tamaño fijo.
- Linux EXT3: es una mejora el EXT2 y tiene discos hasta 4TB. Posee registro por diario. Tiene como ventaja un menor consumo de CPU y es considerado como un sistema más seguro que otros.

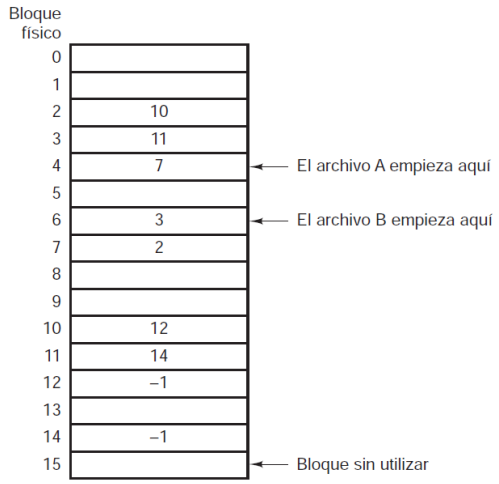
El sistema de archivos que funciona mejor con discos de gran tamaño es el NTFS.

Métodos de asignación de espacio

Aquí podemos encontrar el *contiguo*, donde cada archivo ocupa un conjunto de bloques contiguos en disco. La ventaja es que es sencillo porque solo necesita la localización de comienzo (nº de bloque) y longitud. La desventaja es que no se conoce inicialmente el tamaño, se derrocha espacio y los archivos no pueden crecer a no ser que se compacten.

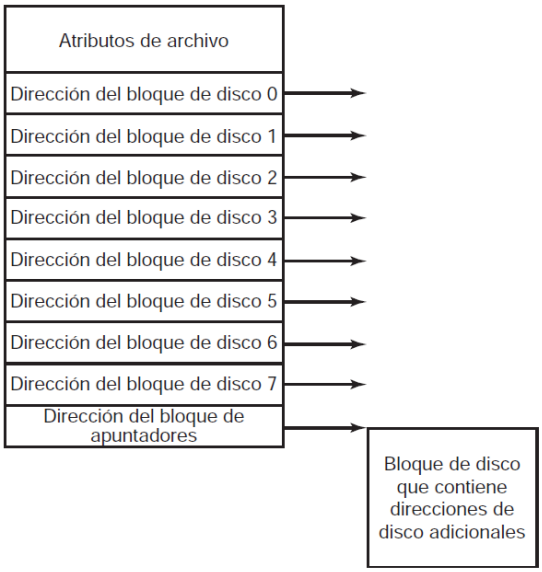
Luego tenemos los *no contiguos o enlazados*, donde cada archivo es una lista enlazada de bloques de disco. Los bloques pueden estar dispersos en el disco. Tiene como ventaja evitar la fragmentación externa, el archivo puede crecer dinámicamente y solo basta almacenar el puntero al primer bloque de archivo. La desventaja es que el acceso directo no es efectivo (si el secuencial), se necesita espacio para los punteros a los enlaces (solución: agrupar en clústers), seguridad por la pérdida de punteros (solución: lista doblemente enlazada).

Luego tenemos una *variación del método enlazado*, que hace frente a las desventajas anteriormente mencionadas, donde tenemos una tabla de asignación de archivos en la memoria principal (FAT – file allocation table) en la cual se reserva una sección del disco al comienzo de la partición para la FAT. Se contiene una entrada por cada bloque del disco y está indexada por número de bloque de disco. Es simple y eficiente siempre que esté en caché. Para localizar un bloque solo se necesita leer en la FAT, optimizando así el acceso directo. En la figura, el archivo A utiliza los bloques 4, 7, 2, 10 y 12, y el archivo B utiliza los bloques 6, 3, 11 y 14. Se puede verificar esto viendo la secuencia y terminando cuando se encuentra el -1 o EOF en otros esquemas.



Con esta organización el bloque completo está disponible para los datos y el acceso aleatorio es mucho más sencillo, sin producir además fragmentación externa. La entrada de directorio necesita mantener solo un entero (el número del bloque inicial) y así luego localizar cada bloque siguiendo la cadena sin importar lo grande que sea el archivo. La principal desventaja de este método es que la tabla debe estar en memoria todo el tiempo para que funcione. La idea de la FAT no se escala muy bien en los discos grandes. Otra desventaja es el posible desperdicio de espacio de los bloques de índice y el tamaño de bloque de índice. Las soluciones son, por ejemplo, bloques índices enlazados, bloques índices multinivel y esquema combinado (en Unix). También tiene como desventaja la fragmentación excesiva de datos, cuando se borran y escriben nuevos archivos, suele dejar fragmentos por todo el soporte de almacenamiento, complicando la lectura y escritura. Para agilizar esta lectura y escritura se utiliza el proceso de desfragmentación que suele ser demasiado largo.

Por último tenemos el método de asignación de que bloques pertenecen a cual archivo: *nodos-i.* Su característica es asociar con cada archivo una estructura de datos (nodo-i o nodo índice), donde la misma lista los atributos y las direcciones de disco de los bloques del archivo. Dado el nodo-i es posible encontrar todos los bloques del archivo.



Tiene como ventaja utilizar una tabla en memoria, el nodo-i necesita estar en memoria sólo cuando está abierto el archivo correspondiente. Si cada nodo-i ocupa N bytes y puede haber un máximo de k archivos abiertos a la vez, la memoria ocupada por el arreglo que contiene los nodos-i para los archivos abiertos es de k*N bytes, hay que reservar este espacio por adelantado. Dentro de las desventajas de esta implementación podemos mencionar que si cada nodo-i tiene un espacio para un número fijo de direcciones de disco, existe un problema cuando el archivo crece más allá del límite. Una solución sería reservar la última dirección de disco, no para un bloque de datos, sino para la dirección de un bloque que contenga más direcciones de bloques de disco, como está en la figura de acá arriba. Ésta implementación es tradicionalmente empleada en Linux.

Gestión de espacios libres

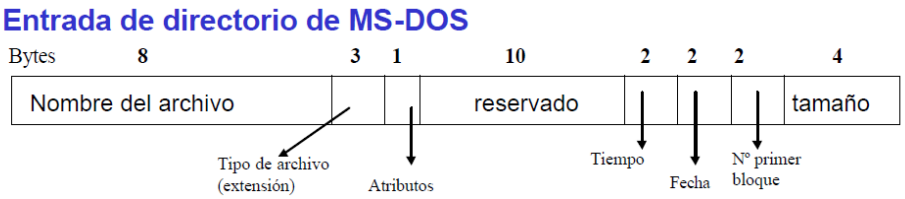
El sistema mantiene una lista de los bloques que están libres: *lista de espacio libre*. La FAT no necesita ningún método. La lista de espacio libre tiene diferentes implementaciones:

- *Mapa o vector de bits*: cada bloque se representa con un bit (0-bloque libre, 1-bloque ocupado), brindando facilidad para encontrar un bloque libre o N bloques libres consecutivos. Es más fácil tener archivos en bloques contiguos. Ineficiente si no se mantiene en memoria principal.
- *Lista enlazada*: enlaza todos los bloques libres del disco, guardando un puntero al primer bloque libre. No derrocha espacio. Es ineficiente porque no es normal atravesar bloques vacíos.
- *Lista enlazada con agrupación*: cada bloque de una lista almacena N-1 direcciones de bloques libres. Obtener muchas direcciones de bloques libres es rápido.
- *Cuenta*: cada entrada de la lista, una dirección de bloque libre y un contador del nº de bloques libres que le sigue.

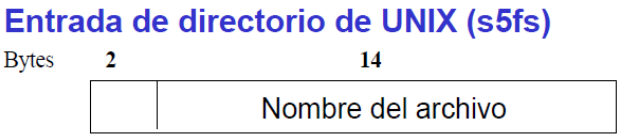
Implementación de directorios

Antes de poder leer un archivo, este debe poder abrirse. Cuando se abre, el SO utiliza el nombre de la ruta para localizar la entrada del directorio, donde ésta entrada provee información necesaria para encontrar los bloques de disco. Existen diferentes casos dependiendo del sistema:

(1) *Nombre de archivo + atributos + dirección de los bloques de datos* (DOS):



(2) *Nombre de archivo + puntero a una estructura de datos que contiene toda la información relativa al archivo* (Unix):



Cuando se abre un archivo, el SO busca en su directorio la entrada correspondiente, extrae sus atributos y la localización de sus bloques de datos y los coloca en una tabla en memoria principal. Cualquier referencia posterior usa la información de dicha tabla.

Posibilidades respecto a la implementación:

- *Lista lineal*: búsqueda lineal en los directorios de principio a fin cuando hay que buscar el nombre de un archivo, es decir, es una lista lineal de nombres de archivos con punteros a los bloques de datos. Sencillo de programar. Es lenta y consume tiempo en las creaciones o búsquedas si no se utiliza una caché software.
- *Tabla Hash*: es una manera de acelerar la búsqueda. Tiene como dificultades el tamaño fijo que posee dicha tabla, necesita previsión para colisiones y su administración es más compleja.

Archivos compartidos

Cuando hay varios usuarios trabajando en conjunto en un proyecto, es necesario compartir los archivos. Es conveniente que aparezca un archivo compartido en forma simultánea en diferentes directorios que pertenezcan a diferentes usuarios. Estos *enlaces* de archivos compartidos pueden ser *simbólicos*, donde se crea una nueva entrada en el directorio, se indica que es de tipo enlace y se almacena el camino de acceso absoluto o relativo del archivo al cual se va a enlazar. Tiene gran número de acceso a disco. El otro enlace o vinculación es el *absoluto (hard)*, donde se crea una nueva entrada en el directorio y se copia la dirección de la estructura de datos con la información del archivo. Brinda problemas a la hora de borrar los enlaces (solución: contador de enlaces).

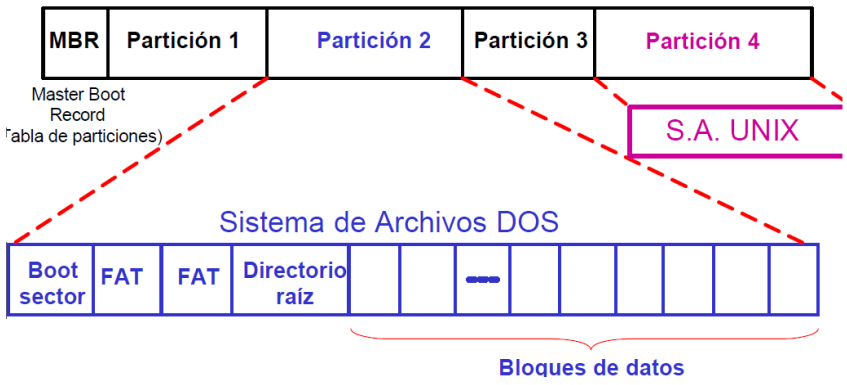
Eficiencia y rendimiento

Los discos suelen ser el principal cuello de botella del rendimiento del sistema. La eficiencia depende de la asignación o alocación en el disco y de la implementación de directorios utilizada. Para proporcionar mejor rendimiento o desempeño encontramos la caché del disco y discos virtuales o discos RAM de almacenamiento temporal.

Recuperación
El sistema debe asegurar que un fallo no genere pérdida o inconsistencia de datos. Existen distintas formas, como el *comprobador de consistencia* que compara los datos de la estructura de directorio con los bloques de datos en disco y trata cualquier inconsistencia. Es más fácil en listas enlazadas que con bloques con índices. Otra forma es usar programas del sistema para realizar copias de seguridad (*backup*) de los datos de disco a otro dispositivo y de recuperación de los archivos perdidos.

Estructura del almacenamiento secundario
Estructura del disco. Desde el punto de vista del SO, el disco se puede ver como un arreglo de bloques (B0, B1, …, Bn). La información se referencia por una dirección formada por varias partes: *unidad* (nº de dispositivo), *superficie* (o cara), *pista* y *sector*. Existe un esquema de asociación de la dirección de un bloque lógico Bi a una dirección física (pista, sector, etc). El área de asignación más pequeña es un bloque.

Gestión del disco
El uso del disco es para contener archivos. El SO necesita registrar sus propias estructuras de datos en el disco. A un mismo disco se le pueden realizar varias *particiones*. Estas particiones se pueden hacer por varios motivos, por ejemplo, para instalar un SO adicional, separar datos, reducir tiempo de desfragmentación, entre otras. A continuación se muestra un ejemplo de cómo se organiza una partición:



El *boot sector* es un bloque de arranque, que es un sector de un disco duro, disquete, o cualquier dispositivo de almacenamiento que contiene un código de arranque. Luego le siguen la *FAT*, el *directorio raíz* y la *zona de datos*.
Para formatear un disco, existen dos posibilidades:

- *Físico*: se ponen los sectores (cabecera y código de corrección de errores) por pista.
- *Lógico*: se escribe la información que el SO necesita para conocer y mantener los contenidos del disco.

Manejo de Espacio de Swapping
Es lo que se conoce como la *memoria virtual*, que usa el espacio del disco como una extensión de la memoria principal. El espacio de *swap* puede ser extraído del sistema de archivos normal o puede estar en una partición separada de disco.

Estructura RAID

Es un conjunto redundante de discos independientes, trabajando cooperativamente, que proveen confiabilidad vía redundancia. Es establecido en 6 niveles diferentes. Los beneficios de un RAID ante un disco duro es que tienen mayor integridad, mayor tolerancia a fallos, mayor rendimiento y capacidad.

SHELL

Los editores, compiladores, enlazadores e intérpretes de comandos no forman parte del SO. El *intérprete de comandos* de UNIX es conocido como SHELL. No forma parte del SO, pero usa muchas características del mismo. Cuando un usuario inicia sesión se inicia un *shell*. Cuando se ejecuta un comando, el *shell* crea un proceso hijo y ejecuta el comando especificado, cuando el hijo termina el *shell* está listo para una nueva entrada.

Llamadas al sistema (system call)

Los SO tienen dos funciones principales: proveer abstracciones a los programas de usuario y administrar los recursos de una computadora. UNIX tiene una llamada al sistema conocida como *read*, con 3 parámetros: uno para especificar el archivo, uno para decir donde se van a colocar los datos (apunta al búfer) y uno para indicar cuantos bytes se deben leer. Para realizar la llamada *read*, el programa llamador primero mete los parámetros en la pila (pasos 1 a 3 de la imagen), donde en el 2do parámetro se pasa por referencia la dirección del búfer, no su contenido.

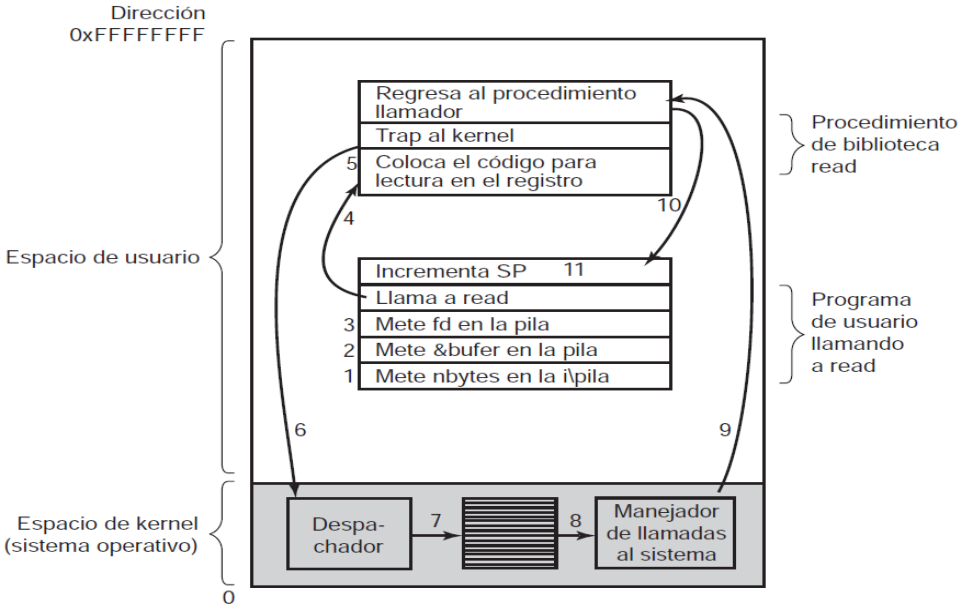


Figura 1-17. Los 11 pasos para realizar la llamada al sistema `read(fd, bufer, nbytes)`.

Luego viene la llamada al parámetro 4, que es la encargada de llamar a todos los procedimientos. En el paso 5 se coloca el número de la llamada al sistema en un lugar en el que el SO lo espera, como en un registro. Después se ejecuta una instrucción TRAP para pasar de modo usuario a modo kernel y empezar la ejecución en una dirección fija. El código kernel que empieza luego del TRAP, examina el número de llamadas al sistema y después la pasa al manejador correspondiente de llamadas al sistema, a través de una tabla de apuntadores a manejadores (paso 7). Se ejecuta el manejador de llamadas al sistema (paso 8) y cuando se termina su trabajo el control se regresa al procedimiento de biblioteca que está en espacio de usuario (paso 9). Luego este procedimiento regresa al programa de usuario en la forma usual que regresan las llamadas a procedimientos (paso 10). Para terminar el programa, el programa de usuario tiene que limpiar la pila (paso 11).

UNIDAD 3: Procesos e hilos de ejecución

Procesos

Un *proceso* es un programa en ejecución (abstracción del SO para identificar un programa en ejecución). Un *programa* es un archivo ejecutable. A cada proceso se le asocia un *espacio de direcciones*, que es una lista de ubicaciones en memoria desde el mínimo hasta el valor máximo permitido, donde el proceso puede leer o escribir información. El espacio de direcciones contiene el programa ejecutable, los datos del programa y su pila. Básicamente un proceso es un recipiente que guarda toda la información necesaria para ejecutar un programa. En muchos sistemas operativos, toda la información acerca de cada proceso se almacena en una tabla conocida como *tabla de procesos*. Un proceso requiere recursos y compite con otros procesos por los mismos recursos.

Entonces los componentes básicos de un proceso son:

- *Espacio de direcciones*: qué espacio de direcciones tiene asociado cada proceso, virtual o físico, y protecciones a memoria.
- *Estado de ejecución*: se detalla el programa que se está ejecutando con sus registros, contadores de programa, punteros a pila, etc.
- *Recursos*: contiene información de los archivos abiertos y directorio. Además de información de scheduling (programación) con el tiempo usado por proceso, prioridades, estado, etc.

Un ejemplo es un proceso llamado *intérprete de comandos* o *shell* que lee comandos en una terminal. El usuario ejecuta un comando para compilar un programa, entonces el *shell* debe crear un proceso para ejecutar el compilador, cuando se termina la compilación ejecuta una llamada al sistema para terminarse a sí mismo. Un proceso puede crear uno o más procesos apartes, denominados *procesos hijos*, y estos a su vez crear procesos hijos. Hay otras llamadas al sistema de procesos para solicitar más memoria (o liberar memoria sin utilizar), esperar a que termine un proceso hijo y superponer su programa con uno distinto.

En cualquier sistema de multiprogramación, la CPU conmuta de un proceso a otro con rapidez, donde estrictamente la CPU está ejecutando sólo un proceso y en 1 segundo podría estar trabajando en varios de ellos, dando la apariencia de paralelismo.

En UNIX los procesos tienen su memoria dividida en 3 segmentos: el *segmento de texto* (código del programa), el *segmento de datos* (variables) y el *segmento de pila*. Donde el segmento de texto crece hacia arriba y la pila hacia abajo.

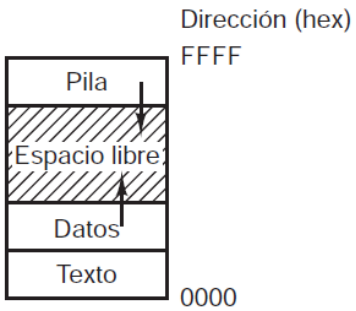


Figura 1-20. Los procesos tienen tres segmentos: de texto, de datos y de pila.

Llamadas al sistema para la administración de procesos

Administración de los procesos FORK: *fork* lo que hace es crear un nuevo proceso, es decir, crea un duplicado exacto del proceso original, incluyendo todos los descriptores de archivos, registros y todo lo demás.

Administración de procesos

Llamada	Descripción
pid = fork()	Crea un proceso hijo, idéntico al padre
pid = waitpid(pid, &statloc, opciones)	Espera a que un hijo termine
s = execve(nombre, argv, entornp)	Reemplaza la imagen del núcleo de un proceso
exit(estado)	Termina la ejecución de un proceso y devuelve el estado

Después de fork, el proceso original y la copia (padre e hijo) se van por caminos separados. Todas las variables tienen valores idénticos al momento de la llamada fork, pero como los datos del padre se copian para crear el hijo, los posteriores cambios en uno de ellos no afectarán al otro. Mediante el uso del PID, los dos procesos pueden ver cuál es el padre y cuál el hijo.
Ahora analizamos la forma en que el shell utiliza el fork. Cuando se escribe un comando en la shell, la misa crea un proceso nuevo usando fork. Este proceso hijo debe ejecutar el comando de usuario. Por ejemplo, ejecuto el comando *cp* para copias: `cp archivo1 archivo2`. Copia el `archivo1` en `archivo2`. El shell se ha bifurcado mediante el fork: el proceso hijo localiza y ejecuta el archivo *cp* y le pasa los archivos de origen y destino al padre.

El modelo del proceso

Todo el software ejecutable en una computadora, donde a veces se incluye al SO, se organiza en varios procesos secuenciales (básicamente procesos). Un *proceso* es una instancia de un programa en ejecución. La conmutación rápida que realiza la CPU de un proceso a otro, se lo conoce como *multiprogramación*. Se considerará como que hay una sola CPU trabajando. Para diferenciar entre *proceso* y *programa*, determinamos que proceso es una actividad de cierto tipo que tiene un programa, una entrada, una salida y un estado. Si un programa se ejecuta por duplicado son dos procesos. El hecho de que dos procesos en ejecución tengan el mismo programa no importa, son procesos distintos.

Creación de un proceso

En SO simples que ejecutan una sola aplicación es posible tener presentes todos los procesos que se vayan a requerir cuando el sistema inicie. En cambio en los SO de propósito general se necesita una forma de crear y terminar procesos.
Hay 4 cuestiones que pueden provocar la *creación de un proceso*:

- El arranque del sistema (boot system).
- La ejecución desde un proceso, de una llamada al sistema para la creación de procesos (acción de otro proceso).
- Una petición de usuario para crear un proceso.
- El inicio de un trabajo por lotes.

Cuando se arranca un SO se crean varios procesos. Algunos de *primer plano*, es decir, los procesos que interactúan con el usuario y realizan trabajos para ellos, mientras que los de *segundo plano* no se asocian con usuarios sino con funciones específicas. Por ejemplo, un proceso en segundo plano que maneja el mail, está inactivo la mayor parte del día pero se activa cuando llega un mensaje. Los procesos que permanecen en segundo plano se conocen como *demonios (daemons)*.

En los sistemas interactivos, los usuarios pueden iniciar un programa escribiendo un comando o haciendo doble clic en un icono, donde se iniciará un proceso y se ejecutará el programa seleccionado. En Linux o Windows los usuarios pueden tener varias ventanas abiertas a la vez, cada una ejecutando un proceso, donde el usuario con el ratón selecciona una ventana e interactúa con el proceso.
Cuando se crea un proceso hay que hacer que un proceso ejecute una llamada al sistema de creación de proceso. Esta llamada indica al SO que cree un proceso y le indica cuál programa debe ejecutar. En UNIX sólo hay una llamada al sistema para crear un proceso: *fork*, donde esta llamada crea un clon exacto del proceso que hizo la llamada. Después de fork, los dos procesos padre e hijo tienen la misma imagen en memoria, las mismas cadenas de entorno y los mismos archivos abiertos. Por lo general el proceso hijo ejecuta después a *execve* o una llamada al sistema similar para cambiar su imagen de memoria y ejecutar un nuevo programa. La familia *exec* sustituye el código del proceso actual con el del hijo. En tanto que en Windows una sola llamada a la función *Win32 (CreateProcess)* maneja la creación de procesos y carga el programa correcto en el nuevo proceso. Tanto en Windows como en Linux, una vez que se crea un proceso, el padre y el hijo tienen sus propios espacios de direcciones distintos. Si cualquier proceso modifica una palabra en su espacio de direcciones no es visible para el otro proceso. En Linux inicialmente el hijo es una copia del padre pero con distintas direcciones, mientras que en Windows los espacios de direcciones de padre e hijo son distintos desde el principio.

```
void main(void) {
        pid_t pid;
        pid = fork();
        if (pid == 0)
                ChildProcess();
        else
                ParentProcess();
}
```

Coordinación de procesos
A través de la llamada *wait()* el padre se mueve a la lista de procesos listos y espera a que alguno de sus hijos termine su ejecución: *pid_t wait(int status)*, o espera que un hijo particular termine: *pid_t waitpid(pid_t pid, int status, int options)*, donde *status* indica la razón de terminación del hijo y *pid* si es mayor a 0 indica un proceso particular, si es igual a -1 a todos los hijos. Si alguno de los hijos termina antes de la invocación de *waitpid* la función retorna inmediatamente, *pid_t* identifica al hijo que ha finalizado y el proceso hijo es eliminado del sistema.

Terminación de procesos
Cuando se crea el proceso, se ejecuta y se realiza el trabajo al que está destinado. Una vez que el proceso termina, puede ser debido a una salida normal (voluntaria), salida por error (voluntaria), error fatal (involuntaria) o eliminado por otro proceso (involuntaria). En Linux es *exit* y en Windows es *ExitProcess*. Una *salida por error* es, por ejemplo, que el usuario ejecute un comando en el que no existe el archivo, entonces el compilador simplemente termina. El *error fatal* se ocasiona debido a un error en el programa, por ejemplo, parte de memoria no existente o división por cero. En cuanto a la *eliminación de un proceso por otro*, en UNIX es el *kill* y en Windows es *TerminateProcess*, donde en ambos casos el proceso eliminador debe tener la autorización necesaria para realizar la eliminación.

Jerarquía de procesos

Cuando un proceso crea a otro, el proceso padre y el proceso hijo continúan asociados en cierta forma. El proceso hijo puede crear por sí mismo más procesos, formando una *jerarquía de procesos*. Un proceso solo tiene un padre. En UNIX, un proceso y todos sus hijos junto con sus descendientes forman un *grupo de procesos*. Windows no tiene el concepto de jerarquía de procesos, todos los procesos son iguales. La única sugerencia de una jerarquía de procesos es que cuando se crea un proceso, el padre recibe un indicador especial *token* llamado *manejador*, que puede utilizar para controlar al hijo. Sin embargo, tiene la libertad de pasar este indicador a otros procesos, invalidando la jerarquía.

Estados de un proceso

A menudo los procesos necesitan interactuar con otros, donde por ejemplo, la salida de uno puede ser la entrada de otro. Cuando un proceso está listo para ejecutarse pero no está disponible aún la entrada que es salida de otro proceso, se debe bloquear el proceso hasta que esté disponible dicha entrada. Un proceso se bloquea porque no puede continuar. También está la posibilidad de que un proceso esté listo y se detenga debido a que el SO ha asignado la CPU a otro proceso por cierto tiempo. Entonces los 3 *estados* que puede tomar un proceso son:

- *En ejecución*: está usando la CPU en ese instante.
- *Listo*: se detuvo temporalmente para que se ejecute otro proceso.
- *Bloqueado*: no puede ejecutarse hasta que ocurra un determinado evento. No depende de la CPU.

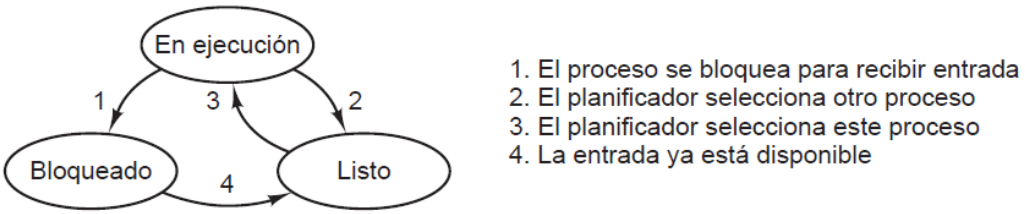


Figura 2-2. Un proceso puede encontrarse en estado "en ejecución", "bloqueado" o "listo". Las transiciones entre estos estados son como se muestran.

Luego tenemos las 4 *transiciones* determinadas en el gráfico entre los 3 estados:

- La transición 1 se da cuando el SO descubre que un proceso no puede continuar en ese momento.
- La transición 2 se da cuando el planificador decide que el proceso en ejecución se ha ejecutado el tiempo suficiente y es momento de dejar que otro proceso use la CPU.
- La transición 3 se da cuando todos los demás procesos han tenido su parte del tiempo en la CPU y es momento de que el primer proceso continúe trabajando en la CPU nuevamente.
- La transición 4 se da cuando se produce el evento externo por el que un proceso estaba esperando (por ejemplo, la llegada del dato de entrada). Si no hay otro proceso ejecutándose ocurre la transición 3 y se ejecutará el proceso, sino tendrá que esperar a que se disponga su turno.

Implementación de los procesos

Para implementar los procesos el SO mantiene una tabla llamada *tabla de procesos*, con sólo una entrada por cada proceso. Estas entradas se llaman *bloques de control de procesos (PCB)*, las cuales contiene información acerca del estado del proceso, incluyendo su contador de programa, puntero a la pila, asignación de memoria, estado de archivos abiertos y todo lo que se tiene que guardar cuando un proceso cambia de estado en ejecución a listo o bloqueado.

Modelación de la multiprogramación

Con la *multiprogramación* el uso de la CPU se puede mejorar. El uso de la CPU se obtiene mediante la fórmula $1-p^n$, donde *p* es la fracción de tiempo que gasta un proceso esperando que se complete una operación de E/S y *n* es la cantidad de procesos en memoria a la vez. Entonces la probabilidad de que todos los *n* procesos estén esperando la E/S es p^n.

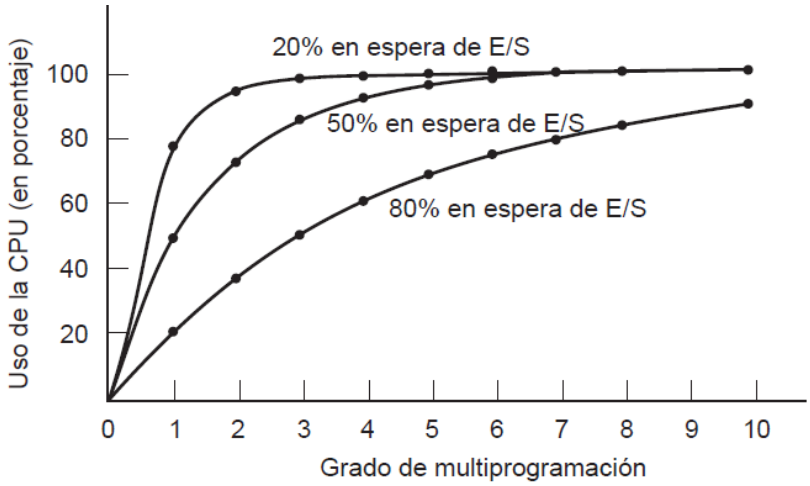


Figura 2-6. Uso de la CPU como una función del número de procesos en memoria.

La figura muestra que si los procesos gastan el 80% del tiempo esperando las operaciones de E/S, por lo menos debe haber 10 procesos en memoria a la vez para que el desperdicio de la CPU esté por debajo del 10%.