Sistemas Operativos

10 de noviembre de 2017

Sistemas Operativos

Del hardware al software

Hemos visto la evolución de los sistemas de cómputo desde el punto de vista del hardware, y cómo llegaron a soportar varios usuarios corriendo varias aplicaciones, todo sobre un mismo equipamiento.

Ahora veremos de qué manera evolucionó el software asociado a esos sistemas de cómputo para permitir que esos diferentes usuarios y esas diferentes aplicaciones pudieran compartir el hardware sin ocasionarse problemas unos a otros, y obteniendo el máximo rendimiento posible del equipamiento.

La pieza que falta en este complejo mecanismo es el **sistema operativo**, un software básico cuya función principal es la de ser intermediario entre los usuarios y el hardware del sistema de cómputo.

Evolución del software de base

Open Shop

Las primeras computadoras estaban dedicadas a una única tarea, perteneciente a un único usuario. Podían ser utilizadas por diferentes usuarios, pero cada uno debía esperar su turno para reprogramarlas manualmente, lo cual era laborioso y se llevaba gran parte del tiempo por el cual esos usuarios pagaban.

Sistemas Batch

Una vez que se popularizaron las máquinas de programa almacenado, se pudo minimizar el tiempo ocioso adoptando **esquemas de carga automática** de trabajos. Un trabajo típico consistía en la compilación y ejecución de un programa, o la carga de un programa compilado más un lote de datos de entrada, y la impresión de

un cierto resultado de salida del programa. Estos trabajos estaban definidos por conjuntos o lotes de tarjetas perforadas, de ahí su nombre de trabajos **por lotes** o, en inglés, *batch*.

Sistemas Multiprogramados

Más adelante, conforme las tecnologías permitían ir aumentando la velocidad de procesamiento, se notó que los procesadores quedaban desaprovechados gran parte del tiempo debido a la inevitable **actividad de entrada/salida**. Así se idearon sistemas que optimizaban la utilización de la CPU, al poderse cargar más de un programa en la memoria y poder conmutar el uso del procesador entre ellos. Éstos fueron los primeros **sistemas multiprogramados**.

Sistemas de Tiempo Compartido

Una vez que llegó la posibilidad de tener varios programas coexistiendo simultáneamente en la memoria, se buscó que la conmutación del uso del procesador entre ellos fuera tan rápida, que pareciera que cada programa funcionaba sin interrupciones. Aunque el sistema era de **tiempo compartido**, el usuario utilizaba la computadora como si estuviera dedicada exclusivamente a correr su programa. Así los sistemas multiprogramados se volvieron **interactivos**.

Computación personal

Todas éstas fueron innovaciones de software, y fueron estableciendo principios y técnicas que serían adoptadas en lo sucesivo. Con la llegada de la computación personal, los sistemas de cómputo eran de capacidades modestas. Los **sistemas operativos** que permitían la ejecución de aplicaciones de los usuarios en estos sistemas de cómputo comenzaron pudiendo correr una sola aplicación por vez y de un solo usuario; es decir, se trataba de sistemas **monotarea** y **monousuario**.

Sin embargo, con la industria de las computadoras personales y la del software para computadoras personales traccionándose una a la otra, aparecieron sistemas operativos **multiusuario** y **multitarea**, sumamente complejos, que se convirtieron en un nuevo terreno para ensayar y mejorar las tecnologías de software y hardware.

Preguntas

- ¿Cuáles son los cinco momentos evolutivos del software de base que reconocemos?
- ¿A qué se llama un **trabajo batch** o lote de trabajo? ¿Qué es un **archivo batch** en el mundo de la computación personal?
- ¿Cuál fue la necesidad que impulsó la creación de sistemas *batch*?

- ¿Cuál fue la necesidad que impulsó la creación de sistemas multiprogramados?
- ¿Cuál fue la necesidad que impulsó la creación de sistemas de tiempo compartido?

Componentes del SO

Los modernos sistemas operativos tienen varios componentes bien diferenciados. Los sistemas operativos **de propósito general** normalmente se presentan en una **distribución** que contiene e integra al menos tres componentes.

- **Kernel**

  El componente que constituye el sistema operativo propiamente dicho es el llamado **núcleo** o **kernel**.

- **Software de sistema**

  Junto al kernel es habitual encontrar un conjunto de **programas utilitarios o software de sistema**, que no es parte del sistema operativo, estrictamente hablando, pero que en general es indispensable para la administración y mantenimiento del sistema.

- **Interfaz de usuario**

  También se encuentra junto a este software del sistema alguna forma de **interfaz de usuario**, que puede ser gráfica o de caracteres. Esta interfaz de usuario se llama en general **shell**, especialmente cuando la interfaz es un procesador de comandos, basado en caracteres, y los comandos se tipean.

Sistemas empotrados o embebidos

Hay algunas excepciones a esta estructura de componentes, por ejemplo, en los sistemas operativos **empotrados** o **embebidos** (*embedded systems*), que están ligados a un dispositivo especial y muy específico, como es el caso de algunos robots, instrumental médico, routers, electrodomésticos avanzados, etc.

Estos sistemas operativos constan de un kernel que tiene la misión de hacer funcionar cierto hardware especial, pero no necesariamente incluyen una interfaz de usuario (porque el usuario no necesita en realidad comunicarse directamente con ellos) o no incluyen software de sistema porque sus usuarios no son quienes se encargan de su mantenimiento.

Aplicaciones

Un típico sistema operativo multipropósito, actual, debe dar soporte entonces a la actividad de una gran variedad de aplicaciones. No solamente a la interfaz de

usuario o procesador de comandos, más el software de sistema incluido, sino también a toda la gama de aplicaciones que desee ejecutar el usuario, como programas de comunicaciones (navegadores, programas de transferencia de archivos, de mensajería); aplicaciones de desarrollo de programas (compiladores, intérpretes de diferentes lenguajes).

Kernel

El **kernel** o núcleo es esencialmente un conjunto de rutinas que permanecen siempre residentes en memoria mientras la computadora está operando. Estas rutinas intervienen en todas las acciones que tengan que ver con la operación del hardware.

Recursos

Los **recursos físicos** del sistema son todos los elementos de hardware que pueden ser de utilidad para el software, como la CPU, la memoria, los discos, los dispositivos de entrada/salida, etc. El kernel funciona no solamente como un mecanismo de administración y control del hardware o conjunto de recursos físicos, sino también de ciertos recursos del sistema que son **lógicos**, como los archivos.

Procesos

El kernel tiene la capacidad de poner en ejecución a los programas que se encuentran almacenados en el sistema. Cuando un programa está en ejecución, lo llamamos un **proceso**. El sistema operativo controla la creación, ejecución y finalización de los procesos.

Llamadas al sistema o system calls

El kernel ofrece su capacidad de control de todos los recursos a los procesos o programas en ejecución, quienes le solicitan determinadas operaciones sobre esos recursos. Por ejemplo, un proceso que necesita utilizar un dispositivo de entrada/salida, o un recurso lógico como un archivo, hace una **petición de servicio, llamada al sistema, o system call**, solicitando un servicio al sistema operativo. El servicio puede tratarse de una operación de lectura, escritura, creación, borrado, etc. El sistema operativo centraliza y coordina estas peticiones de forma que los procesos no interfieran entre sí en el uso de los recursos.

Modo dual de operación

Si los procesos de usuario pudieran utilizar directamente los recursos en cualquier momento y sin coordinación, los resultados podrían ser desastrosos. Por ejemplo,

si dos o más programas quisieran usar la impresora al mismo tiempo, en el papel impreso se vería una mezcla de las salidas de los programas que no serviría a ninguno de ellos.

Como el sistema operativo debe coordinar el acceso de los diferentes procesos a esos recursos, resulta necesario que cuente con alguna forma de imponer conductas y límites a esos usuarios y programas, para evitar que alguno de ellos tome control del sistema en perjuicio de los demás. Para garantizarle este poder por sobre los usuarios, el sistema operativo requiere apoyo del hardware: su código se ejecuta en un modo especial de operación del hardware, el **modo privilegiado** del procesador.

Los modernos procesadores funcionan en lo que llamamos **modo dual** de ejecución, donde el ISA se divide en dos grupos de instrucciones.

- Ciertas instrucciones que controlan el modo de operación de la CPU, el acceso a memoria, o a las unidades de Entrada/Salida, pertenecen al grupo de instrucciones del **modo privilegiado**.
- Un programa de usuario que se está ejecutando funciona en modo **no privilegiado**, donde tiene acceso a la mayoría de las instrucciones del ISA, pero no a las instrucciones del modo privilegiado.

Llamadas al sistema

El procesador ejecutará instrucciones del programa en ejecución en modo no privilegiado hasta que éste necesite un servicio del sistema operativo, tal como el acceso a un recurso físico o lógico.

Para requerir este servicio, el proceso ejecuta una instrucción de **llamada al sistema** o **system call**, que es la única instrucción del conjunto no privilegiado que permite a la CPU conmutar al modo privilegiado.

La llamada al sistema conmuta el modo de la CPU a modo privilegiado **y además** fuerza el salto a una cierta dirección fija de memoria donde existe código del kernel. En esa dirección de memoria existe una rutina de atención de llamadas al sistema, que determina, por el contenido de los registros de la CPU, qué es lo que está solicitando el proceso.

Con estos datos, esa rutina de atención de llamadas al sistema dirigirá el pedido al subsistema del kernel correspondiente, ejecutando siempre en modo privilegiado, y por lo tanto, con completo acceso a los recursos.

El subsistema que corresponda hará las verificaciones necesarias para cumplir el servicio:

- El usuario dueño del proceso, ¿tiene los permisos necesarios?
- El recurso, ¿está disponible o está siendo usado por otro proceso?
- Los argumentos proporcionados por el proceso, ¿son razonables para el servicio que se pide?, etc.

Si se cumplen todos los requisitos, se ejecutará el servicio pedido y luego se volverá a modo no privilegiado, a continuar con la ejecución del proceso.

Ejecución de aplicaciones

Al ejecutar procesos de usuario o de sistema se pone en juego una jerarquía de piezas de software que ocupa varios niveles.

Normalmente, cualquier aplicación que funcione en el sistema, ya sean las del sistema o las generadas por el usuario, competirá con las demás por los recursos en igualdad de condiciones.

Todas las aplicaciones, en algún momento, requieren funciones que ya están preparadas para su uso y almacenadas en **bibliotecas** especializadas en algún área.

Algunas aplicaciones pueden requerir funciones matemáticas; otras, de gráficos; algunas, de comunicaciones. Todas ellas requerirán, sin duda, funciones de entrada/salida. Cada grupo de estas funciones está encapsulado en una o varias bibliotecas que forman parte del sistema.

La **vinculación** de los programas de usuario con las bibliotecas puede hacerse al tiempo de compilación o, cuando las bibliotecas son **de carga dinámica**, al tiempo de ejecución.

Al ejecutarse los procesos, normalmente las bibliotecas necesitan recurrir a servicios del kernel para completar su funcionamiento. Los diferentes subsistemas del kernel se ocupan de cada clase de servicios y de manejar diferentes clases de recursos.

Por ejemplo:

- Si un proceso necesita solicitar más memoria durante la ejecución, la pedirá al subsistema de **gestión de memoria**.
- Cada vez que un proceso escriba datos en un archivo, estará comunicándose, a través de una biblioteca, con el subsistema de **gestión de archivos**.
- Si un proceso necesita enviar o recibir datos a través de la red, el kernel pondrá en funcionamiento el **driver** de la interfaz de red, la pieza de software que sabe comunicarse con ese hardware.

La comunicación entre los procesos de usuario y sus bibliotecas, por un lado, y el kernel y sus subsistemas, por otro, se produce cuando ocurre una llamada al sistema o system call. Es en este momento cuando se cruza el límite entre modo usuario y modo privilegiado, o espacio de usuario y espacio del kernel.

Una cronología de los SO

Entre la década de 1960 y principios del siglo XXI surgieron gran cantidad de innovaciones tecnológicas en el área de sistemas operativos. Muchas de ellas han

tenido éxito más allá de los fines experimentales y han sido adoptadas por sistemas operativos con gran cantidad de usuarios. Diferentes sistemas operativos han influido en el diseño de otros posteriores, creándose así líneas genealógicas de sistemas operativos.

Es interesante seguir el rastro de lo que ocurrió con algunos sistemas importantes a lo largo del tiempo, y ver cómo han ido reconvirtiéndose unos sistemas en otros.

- Por ejemplo, el sistema de archivos diseñado para el sistema operativo CP/M de la empresa Digital, en los años 70, fue adaptado para el MS-DOS de Microsoft, cuya evolución final fue **Windows**.

  Los diseñadores de Windows NT fueron los mismos que construyeron el sistema operativo VMS de los equipos VAX, también de Digital, y aportaron su experiencia. De hecho, muchas características de la gestión de procesos y de entrada/salida de ambos sistemas son idénticas.

- Otra importante línea genealógica es la que relaciona el antiguo Multics, por un lado, con **Unix** y con Linux; y más recientemente, con el sistema para plataformas móviles Android.

  Unix fue el primer sistema operativo escrito casi totalmente en un lenguaje de alto nivel, el **C**, lo cual permitió portarlo a diferentes arquitecturas. Esto le dio un gran impulso y la comunidad científica lo adoptó como el modelo de referencia de los sistemas operativos de tiempo compartido.

  En 1991 **Linus Torvalds** lanzó un proyecto de código abierto dedicado a la construcción de un sistema operativo compatible con Unix pero sin hacer uso de ningún código anteriormente escrito, lo que le permitió liberarlo bajo una licencia libre. La consecuencia es que Linux, su sistema operativo, rápidamente atrajo la atención de centenares de desarrolladores de todo el mundo, que sumaron sus esfuerzos para crear un sistema que fuera completo y disponible libremente.

  Linux puede ser estudiado a fondo porque su código fuente no es secreto, como en el caso de los sistemas operativos propietarios. Esto lo hace ideal, entre otras cosas, para la enseñanza de las Ciencias de la Computación. Esta cualidad de sistema abierto permitió que otras compañías lo emplearan en muchos otros proyectos.

- Otra empresa de productos de computación de notable trayectoria, **Apple**, produjo un sistema operativo para su línea de computadoras personales Macintosh. Su sistema MacOS estaba influenciado por desarrollos de interfaces de usuario gráficas realizadas por otra compañía, Xerox, y también derivó en la creación de un sistema operativo para dispositivos móviles.

- Otros sistemas operativos han cumplido un ciclo con alguna clase de final, al no superar la etapa experimental, haberse transformado definitivamente en otros sistemas, desaparecer del mercado o quedar confinados a cierto nicho de aplicaciones. Algunos, por sus objetivos de diseño, son menos visibles, porque

están destinados a un uso que no es masivo, como es el caso del **sistema de tiempo real** QNX.

Servicios del SO

Después de conocer estas cuestiones generales sobre los sistemas operativos, veremos con un poco más de detalle los diferentes **servicios** provistos por los principales subsistemas de un SO:

- Ejecución de procesos
- Gestión de archivos
- Operaciones de Entrada/Salida
- Gestión de memoria
- Protección

Si bien la discusión que sigue es suficientemente general para comprender básicamente el funcionamiento de cualquier sistema operativo moderno, nos referiremos sobre todo a la manera como se implementan estos subsistemas y servicios en la familia de sistemas **Unix**, que, como hemos dicho, es el modelo de referencia académico para la mayoría de la investigación y desarrollo de sistemas operativos.

Ejecución de procesos

Creación de procesos

¿Cómo se inicia la ejecución de un proceso? Todo proceso es **hijo** de algún otro proceso que lo crea.

Inicialmente, el SO crea una cantidad de procesos de sistema. Uno de ellos es un **shell** o interfaz de usuario. Este proceso sirve para que el usuario pueda comunicarse con el SO y solicitarle la ejecución de otros procesos.

El *shell* puede ser **gráfico**, con una interfaz de ventanas; o **de texto**, con un **intérprete de comandos**. En cualquiera de los dos casos, de una forma u otra, usando el shell el usuario selecciona algún **programa**, que está residiendo en algún medio de almacenamiento como los discos, y pide al SO que lo ejecute.

En respuesta a la petición del usuario, el SO carga ese programa en memoria y pone a la CPU a **ejecutar el código** de ese programa.

Una vez que el programa está en ejecución, decimos que tenemos un nuevo **proceso** activo en el sistema. Este nuevo proceso es un **hijo** del shell, y a su vez puede crear nuevos procesos hijos si es necesario.

Estados de los procesos

Durante su vida, el proceso atravesará diferentes **estados**. Un proceso puede no estar siempre en estado de ejecución (utilizando la CPU), sino que en un momento dado puede pasar a otro estado, quedando transitoriamente suspendido, para dejar que otro proceso utilice la CPU.

Scheduler o planificador

El ciclo de cambios de estado de los procesos es administrado por un componente esencial del SO, el **scheduler** o **planificador**, que lleva el control de qué proceso debe ser el próximo en ejecutarse. El planificador seguirá una estrategia que permita obtener el máximo rendimiento posible de la CPU.

El scheduler o planificador mantiene una cola de procesos que están esperando por la CPU, y elige qué proceso pasar a estado de ejecutando. Al elegir un nuevo proceso para ejecutar, el que estaba ejecutándose cambia de estado hasta que vuelva a tocarle el uso de la CPU.

- Cuando en el sistema coexisten varios procesos activos durante un espacio de tiempo, decimos que esos procesos son **concurrentes**.
- Cuando, además, puede haber más de uno en ejecución en el mismo instante, decimos que son **paralelos** o que su ejecución es paralela.

Ciclo de estados

Ciclo de estados en un sistema multiprogramado

En un sistema **multiprogramado**, varios procesos pueden estar presentes en la memoria del sistema de cómputo. Durante su vida en el sistema, cada proceso atravesará un ciclo de estados.

- Cuando recién se crea un proceso, su estado es **listo**, porque está preparado para recibir la CPU cuando el planificador o scheduler lo disponga.
- En algún momento recibirá la CPU y pasará a estado **ejecutando**.
- En algún momento, el proceso ejecutará la última de sus instrucciones y finalizará. Es posible que su trabajo sea realmente muy breve y que finalice pronto.
- Sin embargo, es mucho más probable que, durante su vida, el proceso requiera servicios del SO (por ejemplo, para operaciones de entrada/salida, como recibir datos por el teclado o por la red, imprimir resultados, etc.).

- Durante estas operaciones de entrada/salida, el proceso no utilizará la CPU para realizar cómputos, sino que deberá esperar el final de este servicio del SO. Como la operación de entrada/salida potencialmente puede demorarse mucho, el sistema lo pone en estado de **espera** hasta que finalice la operación de entrada/salida.
- Mientras tanto, como la CPU ha quedado libre, el SO aprovecha la oportunidad de darle la CPU a algún otro proceso que esté en estado **listo**.
- Cuando finalice una operación de entrada/salida que ha sido requerida por un proceso, este proceso volverá al estado de **listo** y esperará que algún otro proceso libere la CPU para volver a **ejecutando**.
- Al volver desde el estado de **listo** al estado de **ejecutando**, el proceso retomará la ejecución desde la instrucción inmediatamente posterior a la que solicitó el servicio del SO.

Ciclo de estados en un sistema de tiempo compartido

Los sistemas **de tiempo compartido** están diseñados para ser **interactivos**, y tienen la misión de hacer creer a cada usuario que el sistema de cómputo está dedicado exclusivamente a sus procesos. Sin embargo, normalmente existen muchísimos procesos activos simultáneamente en un SO de propósito general.

Para lograr esto el planificador de estos SO debe ser capaz de hacer los cambios de estado con mucha velocidad. El resultado es que los usuarios prácticamente no perciben estos cambios de estado.

En un sistema **de tiempo compartido**, el ciclo de estados de los procesos es similar al del sistema multiprogramado, pero con una importante diferencia.

- El sistema de tiempo compartido tiene la capacidad de **desalojar** a un proceso de la CPU, sin necesidad de esperar a que el proceso solicite un servicio del SO.
- Para esto, el SO define un **quantum** o tiempo máximo de ejecución (típicamente de algunos milisegundos), al cabo del cual el proceso obligatoriamente deberá liberar la CPU.
- El SO, al entregar la CPU a un proceso que pasa de listo a ejecutando, pone en marcha un reloj para medir un quantum de tiempo que pasará ejecutando el proceso.
- Al agotarse el quantum, el SO **interrumpirá** al proceso y le impondrá el estado de **listo**. Al quedar libre la CPU, el siguiente proceso planificado entrará en estado de ejecución.
- Sin embargo, si un proceso decide solicitar un servicio del SO antes de que se agote su quantum, el ciclo continuará de la misma manera que en el sistema multiprogramado, pasando a estado **en espera** hasta que finalice el servicio.

Comparando multiprogramación y *time sharing*

Notemos que los diagramas de estados del **sistema multiprogramado** y del sistema **de tiempo compartido** se diferencian sólo en una transición: la que lleva del estado de **ejecutando** al de **listo** en este último sistema.

- En un sistema multiprogramado, un proceso sólo abandona la CPU cuando ejecuta una petición de servicio al SO.
- En un sistema de tiempo compartido, un proceso abandona la CPU cuando ejecuta una petición de servicio al SO **o bien** cuando se agota su quantum.

Preguntas

- ¿Por qué un proceso que ejecuta una solicitud de entrada/salida no pasa directamente al estado de **listo**?
- ¿En qué radica la diferencia entre el scheduling de un sistema multiprogramado y el de un sistema de tiempo compartido?
- Si en un sistema multiprogramado se ejecuta un programa escrito de forma que **nunca** ejecuta una operación de entrada/salida, ¿liberará alguna vez la CPU durante su vida?
- ¿Y en un sistema de tiempo compartido?

Concurrencia y paralelismo

Si el sistema de tiempo compartido dispone de **una sola unidad de ejecución o CPU**, habrá solamente **un proceso ejecutándose** en cada momento dado, pero muchos procesos podrán desarrollar su vida al mismo tiempo, alternándose en el uso de esa CPU.

- Cuando los procesos coexisten en el sistema simultáneamente pero se alternan en el uso de **una única CPU** decimos que esos procesos son **concurrentes**. Todos están activos en el sistema durante un período de tiempo dado; sin embargo, no hay dos procesos en estado de ejecución en el mismo momento, por lo cual no podemos decir que se ejecutan "simultáneamente".
- Cuando el sistema de cómputo tiene **más de una CPU**, entonces podemos tener dos o más procesos en estado de ejecución **simultáneamente**, y entonces decimos que esos procesos son **paralelos**. Para tener paralelismo, además de concurrencia debemos tener **redundancia de hardware** (es decir, más de una CPU).

Monitorización de procesos

Los sistemas operativos suelen ofrecer herramientas para monitorizar o controlar los procesos del sistema.

Comando top

En Linux, el comando **top** ofrece una vista de los procesos, información acerca de los recursos que están ocupando, y algunas estadísticas globales del sistema.

Es conveniente consultar el manual del comando (man top) para investigar a fondo los significados de cada uno de los datos presentados en pantalla.

Estadísticas globales

En el cuadro superior, **top** muestra:

- El **tiempo** de funcionamiento desde el inicio del sistema
- La cantidad de usuarios activos
- La **carga promedio** (longitud de la cola de procesos listos) medida en tres intervalos de tiempo diferentes
- La cantidad de procesos o tareas en actividad y en diferentes estados
- Las estadísticas de uso de la CPU, contabilizando los tiempos:
- **De usuario**
- **De sistema**
- **De nice** o "de cortesía"
- **Ocioso**
- **De espera**
- Tiempos imputables a **interrupciones**
- Estadísticas de memoria RAM total, usada y libre, y ocupada por buffers de entrada/salida
- Estadísticas de espacio de intercambio o **swap** total, usado y libre.

Estadísticas de procesos

Para cada proceso, el programa top muestra:

- El **PID** o identificación de proceso
- El **usuario dueño** del proceso
- La **prioridad** a la cual está ejecutando y el **valor de nice** o de cortesía
- El tamaño del **espacio virtual** del proceso y los tamaños del **conjunto residente** y **regiones compartidas**
- El **porcentaje de CPU** recibido durante el ciclo de top
- El **porcentaje de memoria del sistema** ocupada
- El **tiempo** de ejecución que lleva el proceso desde su inicio

- El **comando** u orden con la que fue creado el proceso.

El comando top tiene muchas opciones y comandos interactivos. Uno de ellos muestra los datos de sistema desglosados por CPU o unidad de ejecución.

Comandos de procesos

En los sistemas operativos de la familia de Unix encontramos un rico conjunto de comandos de usuario destinados al control de procesos. Algunos interesantes son:

- ps y pstree listan los procesos activos en el sistema
- nice cambia la prioridad de un proceso
- kill envía una señal a un proceso

Interesante

Administración de procesos

Prioridades de ejecución de procesos

Gestión de archivos

Archivos

La información que se guarda en medios de almacenamiento permanente, como los **discos**, se organiza en **archivos**, que son secuencias de bytes. Estos bytes pueden estar codificando cualquier clase de información: texto, código fuente de programas, código ejecutable, multimedia, etc.

Cualquier pieza de información que sea tratable mediante las computadoras puede ser almacenada y comunicada en forma de archivos.

Sistema de archivos

El componente del SO responsable de los servicios relacionados con archivos es el llamado **sistema de archivos** o **filesystem**.

En general, el filesystem no se ocupa de cuál es el contenido de los archivos, o de qué sentido tienen los datos que contienen. Son las aplicaciones quienes tienen conocimiento de cómo interpretar y procesar los datos contenidos en los archivos.

En cambio, el filesystem mantiene información **acerca** de los archivos: en qué bloques del disco están almacenados, qué tamaño tienen, cuándo fueron creados, modificados o accedidos por última vez, qué usuarios tienen permisos para ejecutar qué acciones con cada uno de ellos, etc.

Metadatos

Como todos estos datos son **acerca de los archivos**, y no tienen nada que ver con los datos **contenidos en** los archivos, son llamados **metadatos**. El sistema de archivos o filesystem mantiene tablas y listas de metadatos que describen los archivos contenidos en un medio de almacenamiento.

Directorios

Una característica compartida por la mayoría de los sistemas de archivos es la organización jerárquica de los archivos en estructura de **directorios**. Los directorios son contenedores de archivos (y de otros directorios).

Los directorios han sido llamados, en la metáfora de las interfaces visuales de usuario, **carpetas**.

Varios significados

En rigor de verdad, el nombre de sistema de archivos o filesystem designa varias cosas, relacionadas pero diferentes:

- **Una pieza de software**

  El filesystem es el subsistema o conjunto de rutinas del kernel responsable de la organización de los archivos del sistema. Es un componente de software o módulo del kernel, y como tal, es **código** ejecutable.

- **Un conjunto de metadatos**

  Pero, por otro lado, también hablamos del filesystem como el conjunto de metadatos acerca de los archivos grabados en un medio de almacenamiento. El filesystem en este sentido, es la **información** que describe unos archivos y reside en el mismo medio de almacenamiento que ellos.

- **Un conjunto de características**

  Además, cuando se diseña un sistema de archivos, se lo dota de ciertas capacidades distintivas. Al referirnos al filesystem, podemos estar hablando del **conjunto de características ofrecidas** por alguna implementación en particular de un sistema de archivos.

  - Algunos sistemas de archivos específicos tienen ciertas restricciones en la forma de los nombres de los archivos, y otros no.
  - Algunos permiten la atribución de permisos o identidades de usuario a los archivos, y otros no.
  - Algunos ofrecen servicios como encriptación, compresión, o versionado de archivos.

Árbol de directorios

En los filesystems de tipo Unix, la organización de los directorios es jerárquica y recuerda a un árbol con **raíz** y ramas. Algunos directorios cumplen una función especial en el sistema porque contienen archivos especiales, y por eso tienen nombres establecidos.

- Por ejemplo, el directorio raíz, donde se origina toda la jerarquía de directorios, tiene el nombre especial "/".
- El directorio lib (abreviatura de **library** o biblioteca) contiene bibliotecas de software.
- Los directorios bin, sbin. /usr/bin, etc., contienen archivos ejecutables (a veces llamados **binarios**).

Nombres de archivo y referencias

Los nombres completos, o **referencias absolutas**, de los archivos y directorios se dan indicando cuál es el camino que hay que recorrer, para encontrarlos, **desde la raíz** del sistema de archivos.

Ejemplo

La referencia absoluta para el archivo texto.txt ubicado en el directorio juan, que está dentro del directorio home, que está dentro del directorio raíz, es /home/juan/texto.txt.

Una **referencia relativa**, por otro lado, es una forma de mencionar a un archivo que depende de dónde está situado el proceso o usuario que quiere utilizarlo. Todo proceso, al ejecutarse, tiene una noción de lugar del filesystem donde se encuentra.

- Por ejemplo, el shell de cada usuario funciona dentro del directorio **home** o espacio privado del usuario.
- Éste es el **directorio actual** del proceso shell.
- Puede ser cambiado utilizando el comando cd.
- El comando pwd dice cuál es el directorio actual de un shell.

La referencia relativa de un archivo indica cuál es el camino que hay que recorrer para encontrarlo **desde el directorio actual** del proceso.

Ejemplo

Para el mismo archivo del ejemplo anterior, si el directorio activo del shell es /home/juan, la referencia relativa será simplemente texto.txt.

La referencia es relativa porque, si el proceso cambia de directorio activo, ya no servirá como referencia para ese mismo archivo.

Elementos del sistema de archivos

Particiones

Los medios de almacenamiento se dividen en **particiones** o zonas de almacenamiento. Cada partición puede contener un sistema de archivos, con sus archivos y metadatos.

Bloques

Los **bloques** son las unidades mínimas de almacenamiento que ofrecen los diferentes dispositivos, como los discos. Un **bloque** es como un contenedor de datos que se asigna a un archivo.

Los archivos quedan almacenados, en los discos y en otros medios, como una sucesión de bloques de datos. El filesystem tiene la responsabilidad de mantener la lista de referencias a esos bloques, para poder manipular los archivos como un todo.

Inodos

Los **nodos índice** o **inodos** son estructuras de datos que describen, cada una, un archivo. Los inodos contienen los principales metadatos de cada archivo, excepto el nombre.

En los sistemas de archivos del tipo de Unix, los nombres y los inodos de los archivos están separados. Como consecuencia, un archivo puede tener más de un nombre.

Superblock

El **superblock** grabado en una partición es una estructura de datos compleja donde se mantienen los datos del sistema de archivos. El superblock contiene una lista de bloques libres y una lista de inodos.

Inodos

Cada **inodo** describe a un archivo de datos en un filesystem. El inodo contiene metadatos como:

- El tamaño en bytes del archivo
- La identidad del usuario dueño del archivo y del grupo al cual pertenece el archivo
- El tipo de archivo
- Puede tratarse de un archivo regular (de datos) o de un directorio
- Puede ser un pseudoarchivo, o dispositivo de caracteres o de bloques

- Puede ser un dispositivo de comunicaciones entre procesos, como un **socket**, un **pipe** o tubería, u otros
- Los permisos o modo de acceso
- Los archivos en Unix pueden tener permisos de lectura, de escritura/modificación, o de ejecución
- Esos permisos se especifican en relación al dueño del archivo, al grupo del archivo, o al resto de los usuarios
- La fecha y hora de creación, de última modificación y de último acceso
- La cuenta de *links*, o cantidad de nombres que tiene el archivo
- Los números de bloque que almacenan los datos, o **punteros** a bloques

Metadatos

Los metadatos de cada archivo, contenidos en su inodo, pueden ser consultados y modificados con comandos de usuario.

Ejemplo

El comando ls -l muestra los nombres de los archivos contenidos en un directorio. Para cada archivo, consulta el inodo correspondiente, extrae de él los metadatos del archivo, y los presenta en un listado.

El listado se compone de varios elementos por cada fila, separados por espacios.

```
$ ls -l util
total 60
-rwxr-xr-x 1 oso oso   40 Apr 18 16:54 github
-rw-r--r-- 1 oso oso 1337 May  4 12:11 howto.txt
-rwxrwxr-x 1 oso oso  458 Feb  9 16:28 macro
drwxr-xr-x 2 oso oso 4096 May 26 18:14 prueba
-rwxr-xr-x 1 oso oso   30 Feb  9 16:28 server
```

Tipo de archivo

El primer elemento de cada fila contiene un carácter que indica el tipo del archivo, y los siguientes caracteres indican los permisos asignados al archivo. El tipo indicado por el **guión** es **archivo regular**, y el indicado por la letra **d** es **directorio**. En el ejemplo, todos los archivos son regulares salvo **prueba**, que es un directorio. Otros tipos de archivo tienen otros caracteres indicadores.

Permisos

Los permisos de cada archivo están indicados por los nueve caracteres siguientes hasta el espacio. Para interpretarlos, se separan en tres grupos de tres caracteres. Los primeros tres caracteres indican los permisos que tiene el usuario **dueño** del archivo; los siguientes tres, los permisos para el **grupo** al cual pertenece el archivo; y los últimos tres, los permisos que tienen **otros usuarios**.

Cada grupo de permisos indica si se permite la **lectura**, **escritura**, o **ejecución** del archivo. Una letra **r** en el primer lugar del grupo indica que el archivo puede ser escrito. Una **w** en el segundo lugar, que puede ser escrito o modificado. Una **x** en el tercer lugar, que puede ser ejecutado. Cuando no existen estos permisos, aparece un carácter **guión**.

- Así, el archivo github del ejemplo tiene permisos **rwxr-xr-x**, que se separan en permisos **rwx** para el dueño (el dueño puede leerlo, escribirlo o modificarlo, y ejecutarlo), **r-x** para el grupo (cualquier usuario del grupo puede leerlo y ejecutarlo), y lo mismo para el resto de los usuarios.
- Por el contrario, el archivo howto.txt tiene permisos **rw-r–r–**, que se separan en permisos **rw-** para el dueño (el dueño puede leerlo, escribirlo o modificarlo, pero no ejecutarlo), **r–** para el grupo (cualquier usuario del grupo puede leerlo, pero no modificarlo ni ejecutarlo), y lo mismo para el resto de los usuarios.

Cuenta de links

La cuenta de links es la cantidad de **nombres** que tiene un archivo.

Dueño y grupo

Las columnas tercera y cuarta indican el usuario y grupo de usuarios al cual pertenece el archivo.

Tamaño

Aparece el tamaño en bytes de los archivos regulares.

Fecha y hora

Aparecen la fecha y hora de última modificación de cada archivo.

Nombre

El último dato de cada línea es el nombre del archivo.

Bloques de disco

El SO ve los discos como un vector de bloques o espacios de tamaño fijo. Cada bloque se identifica por su número de posición en el vector, o **dirección de bloque**. Esta dirección es utilizada para todas las operaciones de lectura o escritura en el disco.

- Cuando el SO necesita acceder a un bloque para escribir o leer sus contenidos, envía un mensaje al controlador del disco especificando su dirección.
- Si la operación es de lectura, además indica una dirección de memoria donde desea recibir los datos que el controlador del disco leerá.
- Si la operación es de escritura, indica una dirección de memoria donde están los datos que desea escribir.

Cada vez que un proceso solicita la grabación de datos nuevos en un archivo, el filesystem selecciona un bloque de su lista de bloques libres. Para agregar los datos al archivo, el filesystem quita la dirección del bloque de la lista de libres, la añade al conjunto de bloques ocupados del archivo, y finalmente escribe en ese bloque los contenidos entregados por el proceso.

Recorrer un archivo (para leerlo o para hacer cualquier clase de procesamiento de sus contenidos) implica acceder a todos sus bloques de disco, en el orden en que han sido almacenados esos contenidos. La información para saber qué bloques componen un archivo, y en qué orden, está en el **inodo** del archivo.

El inodo contiene entre sus metadatos una lista de las direcciones de todos los bloques que contienen la información del archivo. Cada una de estas direcciones de bloques se llama un apuntador o **puntero** a bloque.

Como el inodo es una estructura de datos de tamaño fijo, esta lista de punteros tendrá un tamaño máximo. Como, además, los archivos tienen tamaños muy diferentes, se impone un diseño cuidadoso de esta lista de bloques.

- Si se define en el inodo un espacio demasiado pequeño para guardar la lista de punteros a bloques, cada inodo representará archivos con pocos bloques, y así el filesystem no podrá contener archivos grandes.
- Si, al contrario, el espacio en el inodo reservado para guardar la lista es grande, se podrán almacenar archivos de muchos bloques; pero si la mayoría de los archivos del sistema fueran pequeños, se estaría desperdiciando espacio en el superblock.

Para administrar mejor el espacio en el superblock, y para mantener el inodo de un tamaño razonable, esos punteros a bloques se dividen en tres clases: punteros **directos, indirectos y doble-indirectos**.

Punteros directos

Los punteros **directos** son simplemente direcciones de bloques de datos. El filesystem clásico de Unix tiene una cantidad fija de diez punteros directos en el inodo. Si un archivo tiene una cantidad de bytes igual o menor a diez bloques de disco, los punteros directos permiten recorrer el archivo completo.

Punteros indirectos

Si el archivo es más grande, y los diez punteros directos no alcanzan para enumerar los bloques que lo componen, se utilizan **punteros indirectos**. Un puntero indirecto contiene la dirección de un bloque **que a su vez contiene punteros directos**. Hay dos punteros indirectos en el inodo del filesystem Unix clásico.

Punteros doble-indirectos

Si tampoco son suficientes los punteros directos y los indirectos, el inodo del filesystem clásico de Unix contiene un puntero **doble-indirecto**. Es un puntero a un bloque **que a su vez contiene punteros indirectos**.

Esta estrategia de las tres clases de punteros permite que los inodos sean de tamaño reducido pero tengan la capacidad de direccionar una gran cantidad de bloques. Así, el filesystem puede contener archivos de tamaño considerablemente grande y al mismo tiempo conservar el espacio en el superblock.

Preguntas

Supongamos que un disco ha sido formateado de modo de contener 1 TiB de espacio de almacenamiento, y que el tamaño de un bloque de disco sea de 4 KiB. Propongamos las fórmulas necesarias para responder las siguientes preguntas.

1. ¿Qué cantidad **cB** de bloques habrá en el disco?
2. ¿Con cuántos bits **cb** representaremos cada dirección de bloque? Dicho de otra manera, ¿qué cantidad de bits serán necesarios para un puntero a bloque?
3. ¿Qué tamaño máximo de archivo se puede representar con tres punteros directos a bloque?
4. ¿Cuántos punteros a bloque **cp** caben en un bloque indirecto?
5. ¿Y en un doble-indirecto (**cd**)?
6. ¿Qué tamaño máximo de archivo se puede representar con tres punteros directos a bloque y uno indirecto?
7. Si el inodo tiene 10 punteros directos, 2 indirectos y uno doble-indirecto, ¿cuánto espacio (**tp**) ocupa la tabla de punteros dentro del inodo?
8. ¿Cuál es el tamaño máximo **tm** de un archivo según esta configuración del inodo?
9. Si sabemos que el tamaño promedio de un archivo de datos será de 512 MiB, ¿cuántos archivos (**ca**) podrá haber en el disco?
10. ¿Cuántos inodos (**ci**) deberá haber en el superblock entonces?
11. ¿Cuánto espacio ocupará la lista de inodos (**ti**) en el superblock?
12. ¿Cuánto espacio ocupará la lista de bloques libres (**tl**) en el superblock?
13. ¿Cuánto espacio ocupará el superblock (**ts**) en el disco?

Directorios

Notemos que en ningún momento hemos mencionado el **nombre** de los archivos entre los metadatos. En el filesystem de Unix, los nombres de archivo no se encuentran en el superblock, sino en los directorios. Otros sistemas de archivos adoptan otras estrategias.

En Unix, un directorio es simplemente un conjunto de bloques de datos, como los archivos regulares, pero juega un papel especial en el funcionamiento del sistema de archivos, y sus contenidos tienen un formato especial. Los archivos especiales de tipo directorio son archivos de datos que podemos pensar organizados como una tabla de dos columnas. Contienen una lista de entradas con **nombres de archivo** (o **links**) y números de **inodos** que los representan.

Links o nombres de archivo

Al separar el archivo (representado por el inodo), de su nombre **o link** (contenido en el directorio), el filesystem permite que un archivo pueda tener **más de un nombre**.

- En un mismo directorio puede haber dos entradas con nombres diferentes, que apunten al mismo inodo.
- En dos directorios pueden existir sendas entradas con el mismo o diferente nombre, que apunten al mismo inodo.

En estos casos, el archivo podrá ser accedido por cualquiera de esos nombres. Ninguno de ellos es privilegiado o especial.

Si un proceso intenta borrar un archivo, en realidad estará borrando **uno de sus nombres**. La cuenta de links en el inodo se decrementará en 1. Sólo cuando la cuenta de links, o nombres, llegue a cero, se liberarán los bloques de datos del archivo y se devolverán a la lista de bloques libres en el superblock.

Búsqueda de un archivo en el filesystem

Supongamos que un proceso necesita leer los bloques de datos del archivo /etc/group. Deberá entregarle al filesystem el nombre de este archivo para que pueda localizarlo y devolverle esos datos.

Supongamos además que especifica el nombre mediante una referencia absoluta. El filesystem analizará la referencia absoluta que le ha entregado el proceso, descomponiéndola en sus partes componentes y usándola como mapa para llegar al archivo, desde el directorio raíz.

- Para encontrar el archivo, el filesystem lee el inodo 0, que corresponde al **directorio raíz**, y recoge de allí los bloques de datos del directorio raíz.
- Esos bloques de datos contienen nombres de otros archivos y directorios, junto al número del inodo que los representa. De aquí puede extraer el filesystem el número de inodo que representa al directorio /etc. Leerá este inodo de la tabla de inodos, en el superblock, y recuperará del disco los bloques que contienen a ese directorio. Los números de estos bloques están en la tabla de punteros a bloques del inodo.
- Como /etc es un directorio, contendrá una tabla de nombres de archivo y números de inodos. Aquí podrá encontrarse el número de inodo que corresponde a /etc/group.
- Finalmente, leyendo este inodo, el filesystem recorrerá los punteros a bloques devolviendo el contenido del archivo /etc/group.

Gestión de memoria

En un sistema multiprogramado, la memoria debe ser dividida de alguna forma entre los procesos que existen simultáneamente en el sistema. La tarea de controlar qué proceso recibe qué región de memoria, o **gestión de memoria**, es un problema con varias soluciones.

Mapa de memoria

Un programa se compone, como mínimo, de:

- Las instrucciones para el procesador (**código** o **texto** del programa).
- Los **datos** con los que operarán esas instrucciones.

Para que este programa pueda convertirse en un proceso, tanto instrucciones como datos deben estar almacenados en posiciones de **memoria principal**. Solamente de allí pueden ser leídos por el procesador.

Además, los procesos requieren otros espacios de memoria para varios otros usos.

- Por ejemplo, al llamar a una función o rutina, es necesario guardar temporariamente la **dirección de retorno**, que es la dirección de la instrucción a la cual se debe volver una vez terminada la rutina. Estas direcciones se guardan en una zona denominada la **pila** o **stack** del proceso.
- El proceso puede necesitar crear dinámicamente **estructuras de datos** que no existían al momento de carga del programa en memoria. Estos componentes también necesitan ser almacenados en memoria, típicamente en una zona denominada el **heap**.

Todos estos componentes forman lo que a veces se llama **mapa de memoria** de cada proceso, y requieren memoria física.

Espacios de direcciones

Espacio de direcciones físicas

La memoria física del sistema se ve como un arreglo, vector o secuencia ordenada de celdas o posiciones de almacenamiento. Cada posición tiene una **dirección** que es el número con el que se la puede acceder para leer o escribir su contenido. En un sistema de cómputo, el conjunto de direcciones de la memoria física es un **espacio de direcciones físicas**.

Espacio de direcciones lógicas

Al ejecutarse un proceso, las instrucciones que va ejecutando la CPU **referenciarán** a los objetos del mapa de memoria mediante su dirección. Cada vez que la CPU necesite cargar una instrucción para decodificarla, hará una referencia a la dirección donde reside esa instrucción. Cada vez que una instrucción necesite acceder a un dato en memoria, la CPU hará una referencia a su dirección. El conjunto de todas las direcciones de estos objetos forma el **espacio de direcciones lógicas** del proceso.

Ejemplo

En nuestro modelo MCBE, los espacios físico y lógico coinciden. Como sabemos, una instrucción como **01000111** indica que se debe cargar en el acumulador el contenido de la dirección 7.

- Al ejecutarse esta instrucción, el procesador envía el número 7 al sistema de memoria para que éste le entregue el contenido de esa posición. El procesador hace una **referencia** a la dirección 7. Por lo tanto, la dirección 7 pertenece al espacio lógico del programa.
- Además, el sistema de memoria utiliza directamente el número 7 recibido de la CPU como la dirección de memoria que debe devolver. La posición física consultada por el sistema de memoria es exactamente la número 7.

Traducción de direcciones

En un sistema multiprogramado, los programas son cargados en diferentes posiciones del espacio de memoria física. Esto hace que los espacios **lógico y físico** de direcciones de un proceso, en general, no coincidan.

Para que las referencias a direcciones **lógicas** conserven el sentido deseado por el programador, el sistema utiliza alguna forma de **traducción de direcciones**. Las referencias a direcciones generadas por el procesador pertenecerán al espacio lógico del proceso; pero el mecanismo de traducción de direcciones **mapeará** esas direcciones lógicas a las direcciones físicas asignadas.

Unidad de gestión de memoria o MMU

Esta traducción tiene lugar, automáticamente, en el momento en que el procesador emite una dirección hacia el sistema de memoria, y está a cargo de un **componente especial del hardware**. Este componente se llama la **unidad de gestión de memoria** (**MMU, Memory Management Unit**).

- El sistema de memoria recibe únicamente las direcciones **físicas**, traducidas, y "no sabe" que el procesador ha solicitado acceder a una dirección **lógica** diferente.

- Por su parte, el procesador "no sabe" que la dirección física accedida es diferente de la dirección lógica cuyo acceso ha solicitado.

Si el sistema no ofreciera un mecanismo automático de traducción de direcciones, el programador necesitaría saber de antemano en qué dirección va a ser cargado su programa, y debería preparar las referencias a las direcciones de modo de que ambos espacios coincidieran.

Ejemplo

El siguiente programa sencillo en lenguaje ensamblador de MCBE hace referencias a algunas direcciones.

```
00000         LD   DATO
00001         ADD  CANT
00010         ST   SUMA
00011         HLT
00100  DATO:  10
00101  CANT:  1
00110  SUMA:  0
```

En este ejemplo, DATO, CANT y SUMA son las posiciones de memoria 4, 5 y 6. Si este programa se carga en la posición **cero** de la memoria física, los espacios físico y lógico coincidirán. Sin embargo, en un sistema multiprogramado, es posible que el proceso reciba otras posiciones de memoria física.

Por ejemplo, el programa podría haber sido cargado a partir de la dirección 20 de la memoria. Entonces, la posición de memoria física donde residirá el valor SUMA no es la posición 6, sino la 26.

El mecanismo de traducción de direcciones del sistema **deberá sumar el valor base de la memoria** (en este caso, 20) **a todas las referencias a memoria generadas por el procesador** para mantener el funcionamiento deseado. **Sin** traducción de direcciones, la instrucción ST SUMA almacenará el resultado en la posición cuya dirección física es 6... ¡que pertenece a otro proceso!

Asignación de memoria contigua

Uno de los esquemas de asignación de memoria más simples consiste en asignar una región de memoria **contigua** (un conjunto de posiciones de memoria sin interrupciones) a cada proceso.

Si un SO utiliza este esquema de asignación de memoria, establece **particiones** de la memoria, de un tamaño adecuado a los requerimientos de cada proceso. Cuando un proceso termina, su región de memoria se libera y puede ser asignada a un nuevo proceso.

Fragmentación externa

El problema de este esquema es que, a medida que el sistema opere, las regiones que queden libres pueden ser tan pequeñas que un proceso nuevo no pueda obtener una región de tamaño suficiente, **a pesar de que exista memoria libre en cantidad suficiente** en el sistema. Este fenómeno se llama **fragmentación externa**.

Un remedio para la fragmentación externa es la **compactación** de la memoria, es decir, reubicar los procesos que estén ocupando memoria, de manera de que sus regiones sean contiguas entre sí. De esta forma los "huecos" en la memoria se unen y se crean regiones libres contiguas grandes.

El problema con esta solución es que la reubicación de los procesos es muy costosa en tiempo. Mientras el sistema esté compactando la memoria, los procesos que estén siendo reubicados no podrán realizar otra tarea, y el sistema perderá productividad.

Esta clase de cargas extra en tareas administrativas, que quitan capacidad al sistema para atender el trabajo genuino, se llama **sobrecarga** u **overhead**.

Segmentación

Un esquema de asignación que reduce la fragmentación externa es el de **segmentación**. Con este esquema, el mapa de memoria del proceso se divide en trozos de diferentes tamaños, llamados **segmentos**, conteniendo cada uno un conjunto de instrucciones o de datos.

Durante la compilación de un programa fuente, el compilador distribuye los trozos de código y las estructuras de datos en distintos segmentos. Cada segmento tiene un tamaño o **límite**, calculado y especificado por el compilador, y grabado en la cabecera del archivo ejecutable resultante de la compilación.

Al cargar un programa en memoria, el SO destina cada segmento a una determinada **dirección base** física. La dirección base de cada segmento es utilizada por el mecanismo de traducción de direcciones. El dato de tamaño o límite de cada segmento es utilizado para la **protección**, asegurando que las referencias a memoria generadas por la CPU no rebasen los límites de cada segmento. De esta forma, un proceso no puede corromper el espacio de otros.

Este esquema de asignación de memoria reduce, aunque no elimina, la fragmentación externa, ya que los segmentos son más pequeños y reubicables dinámicamente.

Paginación

El esquema de asignación de memoria conocido como **paginación** considera la memoria dividida en regiones del mismo tamaño (**marcos** de memoria), y el espacio lógico de los procesos dividido en regiones (**páginas**) de igual tamaño que los marcos.

Las páginas de los procesos se asignan individualmente a los marcos, una página por vez.

Al contrario que en un sistema de particiones, en un sistema con paginación de memoria los procesos reciben más de una región de memoria o marco. Los marcos asignados a un proceso pueden no ser contiguos.

Fragmentación interna

Bajo este esquema no hay fragmentación externa, porque, si existe espacio libre, siempre será suficiente para alojar al menos una página. Sin embargo, en general, el tamaño del espacio lógico del proceso no es exactamente divisible por el tamaño de la página; por lo tanto, puede haber algún espacio desaprovechado en las páginas asignadas. Esta condición se llama **fragmentación interna**.

Tabla de páginas

Para poder mantener la correspondencia entre marcos de memoria y páginas de los procesos, el SO mantiene una **tabla de páginas** por cada proceso.

- La tabla de páginas de cada proceso dice, para cada página del proceso, qué marco le ha sido asignado, además de otra información de control.
- La tabla de páginas puede contener referencias a marcos compartidos con otros procesos. Esto hace posible la creación de regiones de memoria compartida entre procesos.
- En particular, los marcos de memoria ocupados permanentemente por el kernel pueden aparecer en el mapa de memoria de todos los procesos.

Paginación por demanda

Debido a que los programas no utilizan sino una pequeña parte de su espacio lógico en cada momento (fenómeno llamado **localidad de referencias**), la asignación por paginación tiene una propiedad muy interesante: no es necesario que todas las páginas de un proceso estén en memoria física para que pueda ser ejecutado.

Esto permite la creación de sistemas con **paginación por demanda**, donde las páginas se cargan en memoria a medida que se necesitan. Un proceso puede empezar a ejecutarse apenas esté cargada la primera de sus páginas en memoria, sin necesidad de esperar a que todo su espacio lógico tenga memoria física asignada.

Cada proceso **demanda** al SO la carga de una página de su espacio lógico al espacio físico en el momento en que referencia algún objeto perteneciente a esa página. De esta manera la actividad de entrada/salida desde el disco a la memoria se reduce al mínimo necesario.

Utilizando 1) **paginación por demanda**, 2) agregando algunas características al mecanismo de **traducción de direcciones**, y 3) contando con un espacio de almacenamiento extra en disco para **intercambio de páginas** o **swapping**, se puede implementar un sistema de **memoria virtual**. La mayoría de los SO multipropósito para hardware con MMU utiliza esta técnica.

Memoria virtual

Con memoria virtual, el espacio de direcciones lógicas y el espacio físico se independizan completamente. El espacio lógico puede tener un tamaño completamente diferente del espacio físico. La cantidad de páginas de un proceso ya no se ve limitada por la cantidad de marcos de la memoria física.

- Podemos ejecutar **más procesos** de los que cabrían en memoria física si debiéramos asignar todo el espacio lógico de una vez.
- Los procesos pueden tener un tamaño de espacio lógico **más grande** de lo que permite el tamaño de la memoria física.

En un sistema de memoria virtual, la tabla de páginas mantiene, además de los números de página y de marco asociados, datos de estado sobre la condición de cada página.

- **Bit de validez**

  Indica si la página del proceso tiene memoria física asignada.

- **Bit de modificación**

  Indica si la página ha sido modificada desde que se le asignó memoria física.

El bit de validez de cada página indica si la página tiene o no asignado un marco, y es crucial para el funcionamiento del sistema de memoria virtual. Cuando la CPU genera una referencia a una página no válida, la condición que se produce se llama un **fallo de página (page fault)** y se resuelve asignando un marco, luego de lo cual el proceso puede continuar.

Además de estos bits de validez y modificación, la tabla de páginas contiene datos sobre los permisos asociados con cada página.

El mecanismo de memoria virtual funciona de la siguiente manera:

- Cada dirección virtual tiene un cierto conjunto de bits que determinan el número de página. Los bits restantes determinan el desplazamiento dentro de la página.
- Cuando un proceso emite una referencia a una dirección virtual, la MMU extrae el número de página de la dirección y consulta la entrada correspondiente en la tabla de páginas.

- Si la información de control de la tabla de páginas dice que este acceso no es permitido, la MMU provoca una condición de error que interrumpe el proceso.
- Si el acceso es permitido, la MMU computa la dirección física reemplazando los bits de página por los bits de marco.
- Si el bit de validez está activo, la página ya está en memoria física.
- Si el bit de validez no está activo, ocurre un fallo de página, y se debe asignar un marco. Se elige un marco de una lista de marcos libres, se lo marca como utilizado y se completa la entrada en la tabla de páginas. Los contenidos de la página se traerán del disco.
- La MMU entrega la dirección física requerida al sistema de memoria.
- Si la operación era de escritura, se marca la página como **modificada**.

Reemplazo de páginas

Cuando no existan más marcos libres en memoria para asignar, el SO elegirá una página **víctima** del mismo u otro proceso y la desalojará de la memoria. Aquí es donde se utiliza el bit de **modificación** de la tabla de páginas.

- Si la página víctima no está modificada, simplemente se marca como **no válida** y se reutiliza el marco que ocupaba.
- Si la página víctima está modificada, además de marcarla como no válida, sus contenidos deben guardarse en el **espacio de intercambio o swap**.

Posteriormente, en algún otro momento, el proceso dueño de esta página querrá accederla. La MMU verificará que la página no es válida y disparará una condición de **fallo de página**. La página será traída del espacio de intercambio, en el estado en que se encontraba al ser desalojada, y el proceso podrá proseguir su ejecución.

Ejemplo

Supongamos un sistema donde existen dos procesos activos, con algunas páginas en memoria principal, y una zona de intercambio en disco.

- El proceso P1 tiene asignadas cuatro páginas (de las cuales sólo la página 2 está presente en memoria principal), y P2, dos páginas (ambas presentes). Hay tres marcos libres (M4, M6 y M7) y la zona de intercambio está vacía.
- P1 recibe la CPU y en algún momento ejecuta una instrucción que hace una referencia a una posición dentro de su página 3 (que no está en memoria).
- Ocurre un fallo de página que trae del almacenamiento la página 3 de P1 a un marco libre. La página 3 se marca como válida en la tabla de páginas de P1.
- Enseguida ingresa P3 al sistema y comienza haciendo una referencia a su página 2.

- Como antes, ocurre un fallo de página, se trae la página 2 de P3 del disco, y se copia en un marco libre. Se marca la página 2 como válida y P3 continúa su ejecución haciendo una referencia a una dirección que queda dentro de su página 3.
- Se resuelve como siempre el fallo de página para la página 3 y P3 hace una nueva referencia a memoria, ahora a la página 4.
- Pero ahora la memoria principal ya no tiene marcos libres. Es el momento de elegir una página víctima para desalojarla de la memoria. Si la página menos recientemente usada es la página 2 de P1, es una buena candidata. En caso de que se encuentre modificada desde que fue cargada en memoria, se la copia en la zona de intercambio para no perder esas modificaciones, y se declara libre el marco M2 que ocupaba.
- Se marca como no válida la página que acaba de salir de la memoria principal. La próxima referencia a esta página que haga P1 provocará un nuevo fallo de página.
- Se copia la página que solicitó P3 en el nuevo marco libre, se la marca como válida en la tabla de páginas de P3, y el sistema continúa su operación normalmente.

Notemos que en este ejemplo existen tres procesos cuyos tamaños de espacio lógico miden **4, 5 y 6 páginas**, dando un total de **15 páginas**. Sin embargo, el sistema de cómputo sólo tiene **ocho marcos**.

Sin paginación por demanda y memoria virtual, solamente podría entrar en el sistema uno de los tres procesos. Durante las operaciones de entrada/salida de ese proceso, la CPU quedaría desaprovechada. Además, si alguno de los procesos tuviera un espacio lógico de más de ocho páginas, no podría ser ejecutado.

Con la técnica de memoria virtual, los tres procesos pueden estar activos simultáneamente en el sistema, aumentando la utilización de CPU. Y, si alguno de esos procesos tuviera un espacio lógico de **más de 8 páginas**, el sistema seguiría funcionando del mismo modo.

Preguntas

1. La cantidad de bits de página y la cantidad de bits de marco, ¿deben ser iguales? ¿Qué posibilidades hay, y qué consecuencias tiene cada una?
2. ¿Cuántos marcos tiene el espacio físico de un sistema de cómputo que utiliza memoria virtual, cuyas direcciones físicas codifican el número de marco en cuatro bits?
3. ¿Cuántas páginas tiene el espacio de direcciones lógicas de un proceso si las direcciones codifican el número de página en tres bits?
4. ¿Cuántos procesos como el anterior pueden estar activos en un sistema de cómputo como el anterior?