Sistemas Operativos

Contenido

1 – Introducción

1.1 Aspectos Generales

Sistema Operativo: colección organizada de extensiones software del hardware, consistente en rutinas de control que hacen funcionar un computador y proporcionan un entorno para la ejecución de los programas. Otros programas se apoyan en las facilidades proporcionadas por el **SO** para obtener acceso a recursos del sistema. El **SO** actúa como interfaz entre los usuarios y el hardware de un sistema informático.

El *objetivo principal* de un **SO** es lograr que el sistema informático se use de manera cómoda, y el objetivo secundario es lograr que el hardware del computador se emplee de manera eficiente.

El **SO** debe asegurar el correcto funcionamiento del sistema informático. El **SO** ofrece ciertos servicios a los programas y sus usuarios para facilitar la tarea de programación. Las funciones visibles al usuario de un **SO** están en gran medida determinadas por las necesidades y características del entorno objetivo que el **SO** está destinado a soportar.

1.2 ¿Qué es un SO?

Un sistema informático puede dividirse en cuatro componentes principales:

1 Hardware: proporciona los recursos básicos de computación.
2 Programas de aplicación: definen la forma en que estos recursos se emplean para resolver problemas informáticos de los usuarios.
3 Usuarios: personas, máquinas, otros computadores, empleadores del sistema informático.
4 SO: controla y coordina el uso del hardware entre los diversos programas de aplicación de los distintos usuarios.

Internamente, un **SO** actúa como *gestor de los recursos* del sistema informático, llevando control sobre el estado de cada recurso y decidiendo quién obtiene un recurso, durante cuánto tiempo y cuándo.

El objetivo fundamental de los sistemas informáticos es ejecutar los programas de los usuarios y facilitar la resolución de sus problemas.

No existe una definición universal aceptada de qué forma parte de un **SO** y qué no. Una definición más común es: un **SO** *es el programa que se ejecuta todo el tiempo en el computador (núcleo), siendo programas de aplicación todo lo demás*.

Los **SO** y la *arquitectura del computador* tienen una gran influencia mutua.

Los **SO** intentan incrementar la productividad de un recurso de proceso tal como el hardware del computador, o de los usuarios de los sistemas informáticos.

Un **SO** puede procesar su carga de trabajos de forma serie o concurrente, es decir los recursos del sistema informático pueden estar dedicados a un solo programas hasta que termine, o pueden ser reasignados dinámicamente entre una colección de programas activos en diferentes etapas de ejecución, llamándose *sistema de multiprogramación*.

1.3 Los Primeros sistemas

Los primeros sistemas se programaban en lenguaje de máquina, programación de la *máquina desnuda*.

Los programas para la máquina desnuda se pueden desarrollar traduciendo manualmente secuencias de instrucciones en código binario o alguna otra base. Después se introducen en el computador las instrucciones y los datos mediante interruptores de consola o, tal vez, a través de un teclado hexadecimal. Los programas se arrancan cargando el registro contador de programa con la dirección de memoria de la primera instrucción. Los resultados de la ejecución se obtienen examinando los contenidos de los registros relevantes y de las posiciones de memoria. Los dispositivos de E/S, si los hay, deben ser controlados ejecutando el programa directamente, o sea, leyendo y escribiendo los puertos de E/S relacionados. Es evidente que la productividad era muy baja.

Un aspecto importante de este entorno era su *naturaleza interactiva directa*. El programador también era el operador del sistema informático. La mayoría de los sistemas

utilizaban un esquema de reservaciones o de registro para asignar tiempo de máquina. Si alguien quería usar el computador, acudía a la hoja de registros, buscaba el siguiente tiempo libre de máquina que fuera conveniente y se registraba para usarla.

Sin embargo, se presentaban ciertos problemas. Si se reservaba una hora y en la ejecución del programa ocurría algún error difícil de solucionar y no se podía terminar en esa hora, había que detenerse y rescatar lo que se pudiera. Por otra parte, si todo salía con éxito en menos tiempo del reservado, el resto del tiempo se inutilizaba la CPU.

El siguiente paso evolutivo vino con la llegada de *dispositivos de E/S* (tarjetas perforadas, cintas de papel perforadas) y con los *traductores*. Los programas, codificados ahora en un lenguaje de programación, se traducen a formato ejecutable mediante un programa, tal como *un compilador* o *un intérprete*.

Con el tiempo se desarrollaron software y hardware adicionales, empezaron a popularizarse los *lectores de tarjetas*, *impresoras de líneas* y *cintas magnéticas*, se diseñaron *ensambladores*, *cargadores* y *combinadores* y se crearon *bibliotecas de funciones comunes*.

Las rutinas que efectuaban *operaciones de E/S* tenían una importancia especial. Cada nuevo *dispositivo de E/S* poseía sus propias características, lo que requería una cuidadosa programación. Entonces, para cada uno de ellos se escribía una subrutina especial la cual se denominaba *manejador de dispositivo* y es la que sabe cómo deben usarse los *buffers*, *indicadores*, *registros*, *bits de control* y *bits de estado* para cada dispositivo.

Más tarde aparecieron los *compiladores de Fortran*, *Cobol*, y otros lenguajes. Entonces para realizar un trabajo debía seguirse los siguientes pasos:

1. Cargar la cinta del compilador.
2. Ejecutar la compilación.
3. Descargar la cinta del compilador.
4. Cargar la cinta del ensamblador.
5. Ejecutar el ensamblador.
6. Descargar la cinta del ensamblador.
7. Cargar el programa objeto.
8. Ejecutar el programa.

1.4 Monitor Sencillo

El tiempo de preparación de los trabajos representaba un verdadero problema. La solución tuvo dos facetas. En primer lugar se contrató a un *operador profesional* para manejar el computador, por lo que el programador ya no operaba directamente la máquina. Al momento en que terminaba su trabajo, el operador iniciaba el siguiente, con lo que se eliminaban los periodos de inactividad en los que el computador permanecía reservado pero no se usaba.

Como el operador no depuraba los programas si ocurría algún error se hacía un vuelco de memoria y de registros. Esto permitía seguir con el próximo trabajo, pero dificultaba la tarea de depuración del programador.

El segundo ahorra de tiempo se logró al reducir el período de preparación. Los trabajos con requisitos semejantes se *agrupaban en lotes* y se ejecutaban como un grupo en el computador.

Durante la transición de un trabajo a otro, la CPU permanecía inactiva. Para superar este inconveniente se desarrolló la *secuencia automática de trabajos*. Lo que se deseaba era un procedimiento para transferir automáticamente el control de un trabajo al siguiente. Para este fin se creó un pequeño programa llamado *monitor residente* el cual se encuentra (reside) en memoria.

Inicialmente se invocaba al *monitor residente*, mediante las *tarjetas de* control se le indicaba los trabajos que debía realizar y con qué datos. Entonces transfería el control a un programa. Cuando éste terminaba, devolvía el control al monitor residente, el cual pasaba el control al siguiente programa. Las *tarjetas de control* contienen *instrucciones para indicar al monitor residente cuál es el programa que se ejecutará*. Éstas tarjetas se identificaban con un carácter especial.

El monitor residente tiene varias partes identificables:

- ☐☐Intérprete de tarjetas de control: éste es el responsable de *leer y llevar a cabo* las instrucciones de las tarjetas al momento de ejecución. El intérprete de tarjetas de control invoca a intervalos a *un cargador*.
- ☐☐Cargador: carga en memoria los programas del sistema y las aplicaciones.
- ☐☐Manejadores de E/S: tanto el *Intérprete de tarjetas de control* como el *Cargador* efectúan operaciones de E/S, por lo tanto el Monitor residente cuenta con un conjunto de manejadores de E/S,. Con frecuencia, los programas del sistema y las aplicaciones se combinan con estos manejadores de dispositivos, lo que ofrece continuidad en la operación y además ahorra espacio en memoria y tiempo de programación.

Estos sistemas *Batch* o de *procesamiento por lotes* funcionan bastante bien. El monitor residente proporciona una *secuencia automática de trabajos*.

El *rasgo definitivo* de un sistema batch es la ausencia de interacción entre el usuario y el trabajo mientras éste se ejecuta.

Son muy apropiados para ejecutar grandes trabajos que necesitan poca interacción.

1.5 Operación fuera de línea

Operación fuera de línea: consiste en la interposición fuera de línea de un dispositivo rápido entre uno lento y la CPU.

Surge el problema que mientras se lleva cabo una operación de E/S, la CPU está inactiva, esperando que termine la E/S, y mientras la CPU está trabajando, los dispositivos de E/S están inactivos.

Una solución común fue sustituir los lentos lectores de tarjetas y las lentas impresoras de líneas con las más rápidas unidades de cinta magnética, y en vez de que la CPU leyera directamente de las tarjetas, éstas primero se copiaban a una cinta magnética. Cuando la cinta se llenaba lo suficiente, se desmontaba y se llevaba al computador. Cuando se requería una tarjeta como entrada para un programa, se leía el registro equivalente de la cinta. En forma análoga, la salida se escribía en la cinta y el contenido de ésta se imprimía más tarde. Los lectores de tarjetas y las impresoras de líneas se operaban *fuera de línea* (off line); no lo hacía el computador central.

Se utilizaban dos métodos:

1. Se desarrollaron *dispositivos de propósito especial* (lectores de tarjetas, impresoras de líneas) que enviaban la salida a la cinta magnética o tomaban la entrada de ella, directamente. Estos dispositivos contaban con hardware adicional diseñado específicamente para esa tarea.
2. Se dedicaba un pequeño computador a la tarea de copiar de la cinta y en ella. Este pequeño computador era un *satélite del computador principal*. El *procesamiento satélite* fue uno de los primeros casos de *sistemas informáticos múltiples* que trabajaban en conjunto para mejorar el rendimiento.

La ventaja principal de la *operación fuera de línea* era que el computador principal ya no estaba restringido por la velocidad de los lectores de tarjetas o las impresoras, sino por la velocidad mucho mayor de la cinta magnética. Además no era necesario modificar los programas de aplicación para pasar de una operación de E/S directa a una fuera de línea.

Esta capacidad para ejecutar un programa con diferentes dispositivos de E/S se llama **independencia de dispositivos**, y se obtiene cuando el **SO** es capaz de determinar el dispositivo que realmente usa un programa cuando solicita E/S. Los programas se escriben para usar los *dispositivos lógicos de E/S*. Las tarjetas de control indican la correspondencia entre los dispositivos lógicos y los físicos.

La verdadera mejora al emplear la operación fuera de línea, es la posibilidad de usar *varios sistemas lector-cinta* y *cinta-impresora* con una misma CPU. No obstante ahora hay un mayor retardo para iniciar la ejecución de un trabajo determinado; primero debe leerse la cinta y luego se demora hasta que se lean otros trabajos que la "llenen". Luego hay que rebobinarla y montarla. Por supuesto, esto es razonable para los sistemas de procesamiento *batch*, pues *varios trabajos semejantes se pueden agrupar en una cinta antes de llevarla al computador*.

El procesamiento fuera de línea permite hacer simultáneas las operaciones de la CPU y de E/S ejecutando estas dos acciones en dos máquinas independientes.

Inconveniente: Un lector de cinta no puede escribir en un extremo mientras la CPU lee del otro.

1.6 Almacenamiento temporal (buffering y spooling)

Registro físico: son definidos por la naturaleza de los dispositivos de E/S.
Registros lógicos: son definidos por los programas de aplicación.
La conversión se realiza por Software.

1.6.1 Almacenamiento temporal con buffers

Un buffer mantiene los registros que fueron leídos y no procesados o los procesados pero no sacados.

Buffering: método para hacer simultánea la E/S de un trabajo con su propio procesamiento.

La idea es: después de haber leído los datos, cuando la CPU está lista para comenzar a trabajar con ellos, se ordena al dispositivo que inicie de inmediato la siguiente lectura. En ese momento tanto la CPU como el dispositivo se encuentran ocupado. Con un poco de suerte, cuando la CPU esté lista para el siguiente elemento de datos, el dispositivo de entrada habrá terminado de leerlo. La CPU puede entonces comenzar el procesamiento de los datos recién leídos.

En la práctica, el buffer pocas veces mantiene ocupados todo el tiempo a la CPU y sus dispositivos de E/S, ya que alguno, el dispositivo de E/S o la CPU terminará primero. Si la CPU termina primero, en el peor de los casos, no tendrá que esperar más que en el caso que no tuviera el *buffer*. Sin embargo, si el dispositivo de entrada es continuamente más rápido que la CPU, el *buffer* se llenará y el dispositivo tendrá que esperar.

El manejo de buffers es generalmente una función del **SO**. El monitor residente o los manejadores de dispositivos incluyen buffers del sistema para cada dispositivo de E/S. *Las llamadas de subrutinas al manejador de dispositivo efectuadas por los programas de aplicación (solicitudes de E/S) normalmente sólo provocan una transferencia de o hacia un buffer del sistema. La operación real de E/S ya se ha efectuado o se llevará a cabo más tarde, tan pronto esté disponible el dispositivo.*

Los buffers ayudan sobre todo a amortiguar las variaciones en el tiempo requerido para procesar un registro. Sin embargo, si la CPU es, en promedio, mucho más rápida que el dispositivo de entrada, el uso de buffers tiene poca utilidad. Esta situación ocurre con los trabajos *limitados por E/S*, donde la E/S es muy grande en relación con los cálculos.

El buffer hace simultánea la E/S de un trabajo con su cálculo.

1.6.2 Almacenamiento temporal con spooling

En un sistema con disco, *las tarjetas se leen directamente del lector de tarjetas al disco*. La ubicación del contenido de las tarjetas se anota en una tabla que mantiene el SO. Cuando se ejecuta un trabajo, el SO satisface las solicitudes de entrada del lector de tarjetas, *leyendo del disco*. Análogamente, cuando el trabajo solicita a la impresora que imprima una línea, ésta se copia a un buffer del sistema y se escribe en el disco. Cuando termina el trabajo se lleva a cabo la impresión real.

A esta forma de procesamiento se le llama almacenamiento temporal con spooling (simultaneous peripheral operation on-line).

Con el spooling se utiliza el disco como un buffer de gran tamaño. El spooling también se usa para el procesamiento de datos en instalaciones remotas.

El spooling hace simultánea la E/S de un trabajo con el cálculo do otro. El uso del spooling tiene un efecto beneficioso directo sobre el rendimiento del sistema, a cambio del costo de unas cuantas tablas y espacio en disco, la CPU puede hacer simultáneos los cálculos de un trabajo de E/S de otros.

Además proporciona una estructura de datos muy importante: un *depósito de trabajos* (*Job pool*). Un depósito de trabajo en el disco permite al **SO** seleccionar cuál será el siguiente trabajo por ejecutar. Cuando existen varios trabajos en un dispositivo de

acceso directo, como un disco, es posible la *planificación de trabajos*. El spooling fue la base de la multiprogramación ya que con esta técnica se logró por primera vez la multiprogramación entre rutinas del **SO** y un programa de aplicación.

1.7 Multiprogramación

La multiprogramación aumenta la utilización de la CPU organizando los trabajos de manera que ésta siempre tenga algo que ejecutar.

El concepto es: el **SO** escoge uno de los trabajos del depósito y lo comienza a ejecutar. En algún momento, el trabajo tendrá que esperar que se efectúe cierta tarea como montar una cinta, que se introduzca un mando mediante el teclado, o que termine una operación de E/S. En un sistema no multiprogramado la CPU estaría inactiva. En la multiprogramación, el **SO** cambia a otro trabajo y lo ejecuta cuando ese trabajo tiene que esperar, la CPU cambia a otro, etc. Eventualmente el primer trabajo deja de esperar y recupera la CPU; mientras haya otro trabajo por ejecutar, la CPU nunca estará inactiva.

Para que pueda haber varios programas listos para ejecutarse, el sistema debe conservarlos todos en la memoria al mismo tiempo. Entonces también se necesita algún tipo de administración de memoria, el sistema debe elegir uno de ellos, esta decisión se denomina *planificación de la CPU*.

Si varios programas se *ejecutan concurrentemente*, es necesario limitar la posibilidad de que se interfieran unos a otros en todas las fases del **SO**, incluyendo la *planificación de procesos, el almacenamiento en disco y la administración de memoria*.

1.8 Tiempo Compartido (o multitarea)

Es una extensión lógica de la multiprogramación. La CPU ejecuta diversas tareas alternando entre ellas, pero estos cambios son tan frecuentes que los *usuarios* pueden interactuar con cada programa mientras se ejecuta.

Un sistema informático interactivo ofrece una comunicación en línea entre el usuario y el sistema. Cuando el **SO** termina de ejecutar un mandato, busca el siguiente "*enunciado de control*" no del lector de tarjetas, sino del teclado del usuario. Éste proporciona un *mandato*, espera la respuesta y decide sobre el siguiente mandato según el resultado del anterior. La mayoría de los sistemas cuentan con un *editor interactivo* para introducir programas y un *depurador interactivo* que apoya las tareas de depuración.

Para que los usuarios tengan un acceso cómodo a los datos y al código, deben disponer de un *sistema de archivos en línea*.

Archivo: un archivo es un conjunto de información relacionada que ha sido definido por su creador.

Los trabajos interactivos suelen estar formados por varias acciones breves, donde los resultados del siguiente mandato pueden ser impredecibles. El usuario envía el mandato y espera los resultados. Por ende, el *tiempo de respuesta* debe ser breve.

Los sistemas de *tiempo compartido* se crearon para ofrecer un uso interactivo de los sistemas informáticos a un costo razonable. Un **SO** utiliza *planificación de CPU* y la *multiprogramación* para proporcionar a cada usuario, que tiene su propio programa en memoria, una pequeña porción de un computador de tiempo compartido. La ejecución de un programa típicamente se efectúa durante un breve lapso antes de que termine o efectúe una E/S. Esta puede ser interactiva, y como esta se realiza a ritmo humano, en lugar de permitir que la CPU esté inactiva mientras esto sucede, el **SO** cambiará rápidamente la CPU al programa de otro usuario.

Un **SO** de *tiempo compartido* permite a los diversos usuarios compartir al mismo tiempo el computador.

Al crecer la popularidad del tiempo compartido, los investigadores han tratado de combinar los sistemas batch y de tiempo compartido en uno solo.

Para lograr un tiempo de respuesta razonable hay que intercambiar los trabajos entre la memoria principal y el disco que ahora funciona como almacenamiento de respaldo para la memoria principal. También es necesario proporcionar *administración del disco*. Estos sistemas ofrecen además un *sistema de archivos on-line* y *protección*.

1.9 Sistemas distribuidos

Un **SO** distribuido es una colección de sistemas informáticos autónomos capaces de comunicación y cooperación mediante interconexiones hardware y software. Estos sistemas evolucionaron a partir de las *redes de computadores*, en las que un número de equipos en gran medida independientes están conectados mediante enlaces y protocolos de comunicación.

Un **SO** distribuido gobierna la operación de un sistema informático distribuido y proporciona una abstracción de máquina virtual a sus usuarios. El objetivo clave es la *transparencia*. Idealmente, la distribución de componentes y recursos debería quedar oculta a los usuarios y a los programas de aplicación a menos que éstos demanden explícitamente lo contrario.

La nueva tendencia de los sistemas informáticos es distribuir los cálculos entre varios procesadores. Existen dos esquemas básicos:

1. Sistema fuertemente acoplado: los procesadores comparten la memoria y un reloj, por lo general la comunicación se efectúa a través de la memoria compartida.
2. Sistema débilmente acoplado: los procesadores no comparten ni memoria ni reloj, pues cada uno cuenta con su propia memoria local. Los procesadores se comunican a través de distintas líneas de comunicación, como canales de alta velocidad o líneas telefónicas.

Los **SO** distribuidos proporcionan generalmente medios para la compartición global de los recursos del sistema, tales como la capacidad computacional, los archivos y los dispositivos de E/S. Además de cumplir con los servicios típicos de un **SO** en cada nodo, también facilita el acceso a recursos remotos, la comunicación con procesos remotos y la distribución de los cálculos.

Razones principales para construir un sistema distribuido:

- **Compartimiento de recursos**: un usuario puede utilizar en una instalación los recursos disponibles en otra.
- **Aceleración de los cálculos**: un cálculo determinado puede dividirse en varios subcálculos que se ejecuten concurrentemente. Además, si en un momento dado una instalación está sobrecargada con trabajos, algunos de ellos pueden pasarse a otras instalaciones con poca carga (compartimiento de cargas).
- **Confiabilidad**: si falla una instalación de un sistema distribuido, las restantes pueden, potencialmente, continuar operando.
- **Comunicación**.

1.10 Sistemas en tiempo real

Suele usarse como *dispositivo de control en una aplicación dedicada*. Existen *sensores* que llevan la información al computador, el cual debe analizar los datos y quiá ajustar los controles para modificar las entradas al computador, el cual debe analizar los datos y quizá ajustar los controles para modificar las entradas de los sensores.

Tiene restricciones temporales bien definidas, por lo que el procesamiento debe llevarse a cabo dentro de los límites definidos o el sistema fallará.

Es habitual que se procesen ráfagas de miles de interrupciones por segundo sin perder un solo suceso.

Aparecen procesos explícitos definidos y controlados por el programador, básicamente cada proceso separado está a cargo del manejo de un único suceso externo. El proceso se activa tras la ocurrencia del suceso en cuestión, que con frecuencia viene señalado por una interrupción. La operación multitarea se consigue planificando los procesos para ser ejecutados independientemente unos de otros. Cada proceso tiene asignado un cierto nivel de prioridad que se corresponde con la importancia relativa del suceso al que sirve. Normalmente el proceso es asignado al proceso de máxima prioridad entre todos aquellos que están preparados para ejecutarse. Los procesos de mayor prioridad expropian generalmente la ejecución de los procesos de prioridad inferior. Esta forma de planificación, denominada *planificación expropiativa basada en prioridades*, es utilizada por la mayoría de los sistemas de tiempo real.

Por lo general no hay ningún tipo de almacenamiento secundario o es limitado, y en su lugar los datos se almacenan en RWM o ROM.

La gestión de memoria es comparativamente menos exigente que en otros sistemas de multiprogramación, debido a que muchos procesos residen permanentemente en memoria y hay poco movimiento de programas entre almacenamiento primario y secundario, por otra parte, los procesos tienden a cooperar estrechamente, necesitándose por tanto soporte para separación y compartición de memoria.

La gestión de dispositivos críticos en tiempo, es una de las características principales, además proporcionan formas sofisticadas de gestión de interrupciones y de almacenamiento de E/S, suelen proporcionar *llamadas al sistema* para permitir a los procesos de usuario (programas) conectarse a vectores de interrupción y prestar servicio a los sucesos directamente.

La gestión de archivos, cuando se encuentra, debe proporcionar *protección* y *control de acceso*, aunque su objetivo principal es la *velocidad de acceso*.

Están ausentes las características más avanzadas de los **SO**, ya que tienden a separar al usuario del hardware.

1.11 Sistema monousuario

Al reducirse costos se hizo factible un sistema para un solo usuario.

Las CPU de estos sistemas informáticos han carecido de las características necesarias para proteger al **SO**. Su objetivo fue cambiando con el tiempo, pasando de la maximización de la utilización de la CPU y los periféricos a la *comodidad* y la *rapidez de respuesta a los usuarios*.

2. Estructura de los Sistemas de Computación

2.1 Sistemas basados en interrupciones

Interrupción: una interrupción es un mecanismo utilizado para lograr la coordinación necesaria entre distintas unidades de un sistema y para responder a condiciones específicas que se presentan en el procesador.

Por cada interrupción se debe seleccionar y ejecutar un procedimiento adecuado que implica siempre alguna combinación de hardware y software.

El hardware debe proporcionar al menos la siguiente información:

- ☐☐El tipo de interrupción que se produce
- ☐☐Brindar información sobre (Descriptor de interrupciones – ID) disponible en memoria para ser leída cuando se necesite.

El software debe poder realizar las siguientes operaciones:

- ☐☐Mantener registros del flujo de operaciones de I/O.
- ☐☐Asociar la interrupción con la operación de I/O que la produjo.
- ☐☐Analizar el resultado y de existir excepción, encargarse de ellas.
- ☐☐Volcar los resultados al programa que lo requiera.
- ☐☐Controlar si hay más operaciones de I/O listas e iniciarlas.

Los ejemplos de interrupciones que pueden presentarse son:

- ☐☐Por mal funcionamiento del hardware. *Hard*
- ☐☐Por finalización de I/O. *Hard*
- ☐☐Por reloj interno (time-out). *Hard*
- ☐☐Por teclado (interrupción externa). *Hard*
- ☐☐Por inicio de I/O. *Soft*
- ☐☐Por *"llamadas"* voluntarias al SO. *Soft*

La ocurrencia de un evento cambia la secuencia de operaciones de la CPU.

(Diagrama de interrupciones)

En la estructura hay un *vector de estado* compuesto por *registros de hardware del procesador* que representan *el estado del programa que en ese momento "corre"* (tiene tomada la CPU). Los registros se identifican:

- ☺☐**AC** y **MQ**, contienen resultados de operaciones aritméticas.
- ☺☐**IX 1**, es un registro índice del procesador.
- ☺☐**Reg. Base** apunta a la más baja dirección de memoria real que puede referenciar el programa que en ese momento "corre".
- ☺☐**IC** el contador de instrucciones, contiene la dirección de memoria de la próxima instrucción a ejecutarse.

También está el *bloque de definición de interrupciones* (IDB) que reside en memoria real en posiciones de memoria de 0 a 27, las primeras 12 (de 0 a 11) contienen 4 entradas de 3 palabras cada una y *cada entrada está asociada a un determinado tipo de interrupción*.

Cada una de estas entradas contiene 3 palabras: JMP, ID Reg. Base

- ☺☐**JMP** apunta a la dirección de memoria real que contiene la primera instrucción de la rutina que atiende la interrupción.
- ☺☐**ID** define la interrupción con información que fue "colocada" por el dispositivo que ocasiona la interrupción.
- ☺☐**Reg. Base** apunta a la menor dirección de memoria real que puede referenciar la rutina que atiende la interrupción.

En el IDB también se observa en la palabra 12 al **SPR** que contiene el puntero a la *próxima área "save"* disponible.

Cuando se produce una interrupción, el *primer paso* es guardar el vector estado en el área de salvado en el lugar indicado por **SPR**, el *segundo paso* es cargar los registros del vector de estado **IC**, **MQ** y **Reg Base** con los contenidos de las palabras **JMP**, **ID** y

Reg. Base extraídos del IDB según el tipo de interrupción que corresponda (si es de procesador o de direccionamiento, los contenidos son colocados inmediatamente sin esperar que se complete la instrucción en curso pues es abortada, en cambio si son por finalización de E/S por Canal se espera a que se complete la instrucción en curso). Una vez procesada la interrupción, el *tercer paso* es reiniciar el programa interrumpido a través de la ejecución de la instrucción **JMP STACK** que por medio de software *transfiere el contenido del área "save" a los registros del vector de estado.*

Para que las operaciones de la CPU y de E/S puedan superponerse se debe contar con un mecanismo que permita realizar la *desincronización* y la *resincronización* de las operaciones. Existen dos métodos (algunos sistemas emplean ambos):

1. **Transferencia de datos por interrupciones:** cada uno de los controladores de dispositivos se encarga de un tipo específico de dispositivo. Un controlador de dispositivos cuenta con almacenamiento local en buffer y un conjunto de registros de propósito especial. El controlador del dispositivo es el responsable de transferir los datos entre el periférico que controla y su buffer local.

   Para iniciar una operación de E/S, la CPU carga los registros apropiados del controlador del dispositivo y luego continúa con su operación normal. Una vez concluida la transferencia de datos, el controlador del dispositivo informa a la CPU que su operación ha terminado, mediante una interrupción.

   Cuando se interrumpe a la CPU, ésta suspende lo que estaba haciendo y transfiere de inmediato a la ejecución a una posición fija, la cual generalmente contiene la dirección de inicio donde se encuentra la *rutina de procesamiento de la interrupción*. Esta rutina transfiere los datos del buffer local del controlador del dispositivo a la memoria principal, luego, la CPU puede continuar con los cálculos interrumpidos. *De esta manera, los dispositivos de E/S y la CPU pueden operar concurrentemente*.

   La arquitectura de interrupciones también debe guardar la dirección de la instrucción interrumpida, y quizás también, otros registros como *acumuladores* o *registros de índices*, para luego recuperarlos. Las arquitecturas más recientes almacenan la dirección de retorno en la pila del sistema.

   Usualmente las interrupciones se desactivan mientras se procesa una.
2. **Transferencia de datos por Acceso Directo a Memoria (DMA):** después de fijar los buffers, punteros y contadores para los dispositivos de E/S, el *controlador del dispositivo transfiere directamente todo un bloque de datos de su almacenamiento en buffer a la memoria, o viceversa, sin que intervenga la CPU, solo se genera una interrupción por bloque*. El controlador de DMA tiene sus registros con las direcciones fuente y destino apropiadas y la longitud de la transferencia. Esto lo lleva a cabo un manejador de dispositivo, que sabe exactamente cómo proporcionar esta información al controlador. Luego se ordena al controlador de DMA (por medio de bits de control en un registro de control) que comience la operación de E/S. El controlador del DMA interrumpe a la CPU cuando termina la transferencia.

2.2 Estructura de E/S

Los sucesos casi siempre se avisan mediante una interrupción, o una trampa.

Trampa: interrupción generada por el software a causa de un error.

Un **SO** está dirigido por interrupciones. La naturaleza dirigida por interrupciones de un **SO** define su estructura general.

Una de las principales clases de sucesos que debe manejar el **SO** son las *interrupciones de E/S*. Un dispositivo de E/S generará una *interrupción al terminar una solicitud de E/S*, en esta situación hay dos posibilidades:

1. El caso más simple, la E/S se inicia y al terminar, se devuelve el control al programa de usuario.
2. Devolver el control al programa de usuario sin esperar que la E/S termine. La espera de la terminación de la E/S puede lograrse de dos maneras. Algunos computadores tienen una instrucción especial de espera (*wait*). Las máquinas

que no poseen esta instrucción pueden tener un *ciclo de espera* (no recomendado).

Una gran ventaja de esperar siempre a que termine la E/S es que como máximo hay una solicitud de E/S pendiente en cada ocasión.

Una alternativa para procesar varias E/S es comenzar la E/S y devolver de inmediato el control al programa usuario. Se requiere entonces una *llamada al sistema* (una solicitud al **SO**) para permitir al usuario que espere la terminación de la E/S, por tanto, aún necesitamos el anterior código de espera, y también debemos llevar un control de varias solicitudes de E/S, por tanto, aún necesitamos el anterior código de espera, y también debemos llevar un control de varias solicitudes de E/S al mismo tiempo. Con este fin, el **SO** usa una tabla que contiene una entrada por cada dispositivo de E/S: la *tabla de estado de dispositivos*.

Cada entrada de la tabla indica el tipo de dispositivos, así como su dirección y su estado (no funciona, inactivo u ocupado). Puesto que es posible efectuar varias solicitudes para el mismo dispositivo, podemos tener una lista o cadena de solicitudes en espera; Así, además de la tabla de dispositivos de E/S, un **SO** puede contar con una lista de solicitudes para cada dispositivo.

Un dispositivo de E/S interrumpe cuando requiere servicio. Dada esta situación, el **SO** primero determina cuál fue el dispositivo de E/S que causó la interrupción. Luego acude, usando un índice, a la tabla de dispositivos de E/S para determinar el estado del dispositivo y modificar la entrada de la tabla para reflejar la aparición de la interrupción. Para la mayoría de los dispositivos, una interrupción indica la terminación de una solicitud de E/S. Si está en espera alguna otra solicitud para ese dispositivo, el **SO** comienza a procesarla.

2.3 Operación de modo dual

Con el compartimiento, un error de programa podría afectar (adversamente) a varios trabajos.

Un **SO** correctamente diseñado debe asegurar que un programa incorrecto (o malintencionado) no provoque la ejecución incorrecta de otros programas.

El hardware detecta muchos errores de programación. Si el programa de usuario falla de alguna manera, el hardware dirigirá una *trampa* al **SO.** La trampa transfiere el control al **SO** a través del vector de interrupciones entonces, el **SO** debe terminarlo anormalmente. Se envía un mensaje de error apropiado y se vuelca la memoria del programa.

Este método funciona bien siempre que el hardware detecte el error, pero para detectar todos los errores, se requiere protección para cualquier recurso del compartido. La estrategia que se utiliza es proporcionar apoyo del hardware que permita distinguir entre diversos modos de ejecución. Por lo menos necesitamos *dos modos de operación* distintos: *modo usuario* y *modo monitor*. Para esto se añade un bit, llamado *bit de modo* al hardware del computador. Ahora se puede distinguir entre una ejecución efectuada por el **SO** y una efectuada por el usuario.

El *modo dual de operación* nos proporciona un medio para proteger al **SO** de los usuarios errantes, y a éstos de ellos mismos. Logramos esta protección clasificando algunas de las interrupciones de la máquina que pueden causar daño como *instrucciones privilegiadas*, a las cuales el hardware permite ejecutar solo en modo monitor.

2.4 Hardware de protección

Un programa de usuario puede alterar el funcionamiento normal del sistema produciendo instrucciones de E/S ilegales, teniendo acceso a posiciones de memoria dentro del **SO** mismo o rehusándose a liberar la CPU.

Para evitar que un usuario lleve a cabo una E/S ilegal, todas las instrucciones de E/S se definen como privilegiadas, así los usuarios no pueden ejecutar directamente instrucciones de E/S, tienen que hacerlo a través del **SO**. También hay que proteger las rutinas de procesamiento de interrupción en el **SO**. Esta protección debe ser proporcionada por el hardware.

2.4.1 Mecanismo de direccionamiento

El **SO** deberá determinar cómo las direcciones de los programas son asociados con sus ubicacioners en memoria, como se formarán las ubicaciones de las distintas posiciones y como se estructurarán los programas de aplicación para una máquina dada.

Lo que se necesita para separar el espacio de memoria de cada programa es la capacidad de determinar el intervalo de direcciones legales a las que puede tener acceso un programa, y proteger la memoria fuera de ese espacio. Esta protección se puede proporcionar con dos registros, uno *base* y otro *límite*.

El hardware de la CPU logra esta protección comparando *cada una* de las direcciones generadas en modo usuario con los registros.

El **SO** pude cargar valores en los registros base y límite mediante una instrucción privilegiada especial.

Cuando existe reasignación de memoria, es posible combinar mecanismos de direccionamiento y protección.

2.4.2 Mecanismo de protección de la CPU

Para asegurar que el **SO** mantenga el control, debemos evitar que un programa de usuario entre en un ciclo infinito y nunca devuelva el control al **SO**. Para lograrlo, podemos usar un *cronómetro*, que puede fijarse para interrumpir al computador después de determinado periodo. Si el cronómetro interrumpe, el control se transfiere automáticamente al **SO**, el cual puede tratar a la interrupción como un error fatal, o decidir que debe darse mayor tiempo al programa.

Una técnica sencilla consiste en asignar al contador un valor inicial que corresponda a la cantidad de tiempo permitida para la ejecución del programa.

El uso más común del cronómetro es implantar el tiempo compartido.

Otro uso del cronómetro es calcular la hora actual, pero las maquinas de ahora constan hardware separado para eso.

2.5 Ámbito de procesamiento

(Diagrama de ámbito de procesamiento)

El programa "ve" las instrucciones de trabajo (instrucciones del procesador, UAL) y su área de memoria direccionable definida de 0 a X, también "ve" una serie de módulos del **SO** que actúan como interface entre el programa usuario y las instrucciones privilegiadas que proveen servicios tal como si fueran instrucciones de máquinas ejecutables.

Lo que el programa "no ve" son las instrucciones privilegiadas que están ocultas y un área de memoria protegida también oculta, donde se almacenan las rutinas del **SO** y el resto de los programas usuarios concurrentes.

Una instrucción cualquiera del programa usuario escrita en lenguaje de alto nivel, en tiempo de compilación puede ser traducida a un conjunto de instrucciones de trabajo elementales del hardware de base, o a una llamada al monitor. Si se trata de una instrucción de E/S, será traducida a una llamada al monitor y las razones son las siguientes: como la E/S corre asincrónicamente con el programa o hace que el programa pierda el control de la CPU, algún mecanismo de esta arquitectura extendida del hardware debe también realizar la coordinación del programa con su propia E/S.

En este ámbito de multiprogramación y de recursos compartidos no se permite que los compiladores generen todas las funciones dentro del programa compilado. Muchos programas necesitan ejecutar idénticas funciones, si se definen mecanismos que permitan a los programas compartir el mismo código, la memoria será suficientemente usada, disminuyendo el espacio requerido para funciones duplicadas a expensas de mecanismos de coordinación.

2.6 Arquitectura general del sistema

La mayoría de los sistemas de computación modernos cuentan con una instrucción especial denominada *llamada al sistema*.

Cuando se ejecuta, el hardware la trata como una interrupción. El control pasa a través del vector de interrupciones a una rutina de procesamiento del **SO** y el bit de modo

se coloca en modo monitor. La rutina de procesamiento de la llamada al sistema forma parte del **SO**; el monitor examina la instrucción que provoca la interrupción para determinar que ha ocurrido una llamada al sistema. Un parámetro indica qué tipo de servicio requiere el programa de usuario, y la información adicional necesaria para la solicitud puede pasarse en registros o en memoria (pasando en registros los punteros a las posiciones de memoria). El monitor ejecuta la solicitud y devuelve el control a la instrucción siguiente a la de la llamada al sistema.

Cuando se realiza una operación de E/S, el programa usuario ejecuta una llamada al sistema, y el **SO**, que se ejecuta en modo monitor, revisa la validez de la solicitud y, si lo es, realiza la E/S solicitada. El **SO** devuelve entonces el control al usuario.

2.7 Diferentes clases de computadoras

2.7.1 Sistemas multiprocesadores

Éstos sistemas tienen más de una CPU en estrecha comunicación, compartiendo el canal del computador y en ocasiones la memoria y los dispositivos periféricos.

Estos sistemas incrementan la productividad, aunque la relación de aceleración de *n* procesadores, no es *n*, sino menor que *n*.

La confiabilidad es otra ventaja. Si las funciones se pueden distribuir adecuadamente entre los diversos procesadores, *entonces la avería de un procesador no detendrá el sistema*, únicamente lo hará más lento. Esta capacidad se llama *degradación progresiva*, a los sistemas que lo implementean, *fallas suaves*.

La operación continúa en presencia de errores y requiere un mecanismo que permita detectar, diagnosticar y corregir (de ser posible) el error. El sistema *Tandem* emplea una réplica de hardware y de software, al fallo de la principal, se utiliza la secundaria.

Algunos sistemas usan el *multiprocesamiento asimétrico*. Un procesador principal controla el sistema y los demás acuden a él para recibir instrucciones o tienen tareas predefinidas. Este esquema define una relación *amo-esclavo*, en la que el procesador amo planifica y asigna trabajo a los procesadores esclavos. Dado que las CPU están separadas, una puede estar inactiva mientras otra está sobrecargada, entonces, para que no pase esto, los procesadores pueden compartir ciertas estructuras de datos. Este tipo de multiprocesamiento es más común en sistemas de gran tamaño.

En los sistemas antiguos se usaban procesadores más pequeños, para manejar los lectores de tarjetas y las impresoras de líneas y efectuar la transferencia entre ellos y el computador principal. A estos lugares se los llamaban instalaciones de entrada remota de trabajos (RJE). En un sistema de tiempo compartido, la mayoría de los sistemas cuentan con un procesador frontal separado que maneja toda la E/S de terminales. Estos sistemas adolecen una reducción de la confiabilidad por el aumento de la especialización.

2.7.2 Computadoras personales

Algunos computadores personales incluyen el procesamiento multitarea.

Ahora la utilización de la CPU ya no es un factor primordial.

Muchos de los **SO** de computadoras personales son derivados de **SO** de sistemas más grandes.

3. Estructura de los Sistemas Operativos

3.1 Componentes del Sistema

Un **SO** proporciona el entorno dentro del cual se ejecutan los programas. Para construir este entorno, dividimos lógicamente al **SO** en pequeños módulos y creamos una interfaz bien definida para estos programas.

3.1.1 Administración de procesos

Un proceso es un programa en ejecución (Ej.: un trabajo por lotes, un programa de usuario en tiempo compartido, una tarea del sistema).

En general un proceso necesita recursos. Estos recursos se proporcionan al crear el proceso, o se le asignan mientras se ejecutan. Además de los recursos físicos y lógicos, se pueden transferir algunos datos iniciales a un proceso a través de un parámetro.

Un programa es una entidad pasiva.

Un proceso es una entidad activa.

La ejecución de un proceso debe efectuarse secuencialmente, es decir, en cualquier momento se ejecuta como máximo una instrucción del proceso. Los programas, frecuentemente, generan varios procesos durante su ejecución.

Un proceso es la unidad de trabajo de un sistema. Dicho sistema consiste en un conjunto de procesos, algunos de los cuales son procesos del **SO**. Potencialmente, todos estos procesos pueden ejecutarse en forma concurrente, *multiplexando la CPU* entre ellos.

El **SO** es responsable de las siguientes actividades relacionadas con la *administración de procesos*:

- ☺☐**Crear y eliminar procesos de usuarios y sistema.**
- ☺☐**Suspender y reanudar la ejecución de los procesos.**
- ☺☐**Proporcionar mecanismos para la sincronización de procesos.**
- ☺☐**Proporcionar mecanismos para la comunicación de procesos.**
- ☺☐**Proporcionar mecanismos para el manejo de bloques mutuos.**

3.1.2 Administración de la memoria principal

La memoria es un gran arreglo de palabras, cada una con su propia dirección.

En la memoria residen los datos y programas, que *lee* y *escribe* la CPU y los dispositivos de E/S (mediante DMA).

El **SO** es responsable de las siguientes actividades relacionadas con *la administración de memoria*:

- ☺☐**Llevar un control de cuáles son las zonas de la memoria que se están usando y quién las usa.**
- ☺☐**Decidir qué procesos se cargarán en memoria cuando haya espacio disponible.**
- ☺☐**Asignar y recuperar el espacio en memoria según se requiera.**

3.1.3 Administración de almacenamiento secundario

Muchos programas residen (cuando no están siendo utilizados) en un medio de almacenamiento secundario, incluyendo el **SO**. Por eso una buena administración de este medio es vital.

El **SO** es responsable de las siguientes actividades relacionadas con *la administración de discos*:

- ☺☐**Administración del espacio libre.**
- ☺☐**Asignación de almacenamiento.**
- ☺☐**Planificación de las operaciones sobre el disco.**

3.1.4 Administración del sistema de E/S

*Uno de los objetivos de un **SO** es ocultar al usuario las particularidades de los dispositivos de hardware.*

El *sistema de E/S del* **SO** consiste en:

- ☺☐**Un sistema de memoria *caché* mediante *buffering*.**

☺☐**Una interfaz general con los manejadores de dispositivos.**
☺☐**Manejadores para dispositivos de hardware específicos.**

Solo el manejador del dispositivo conoce las particularidades del dispositivo al cual está asignado.

3.1.5 Administración de archivos

La administración de archivos es uno de los componentes más visibles de un SO.

Al haber muchos dispositivos diferentes (ópticos, magnéticos, etc.), el **SO** ofrece una perspectiva lógica uniforme del almacenamiento de información.

El **SO** es responsable de las siguientes actividades relacionadas con *la administración de archivos*:

☺☐**Creación y eliminación de archivos.**
☺☐**La creación y eliminación de directorios.**
☺☐**El manejo de operaciones primitivas para manipular archivos y directorios.**
☺☐**La correspondencia entre archivos y almacenamiento secundario.**
☺☐**La copia de seguridad de archivos en medios de almacenamiento no volátiles.**

3.1.6 Sistema de protección

Los distintos procesos de un **SO** deben ser protegidos unos de otros. Con este fin se proporcionan mecanismos para asegurar que los recursos puedan ser usados únicamente por aquellos procesos que han recibido la debida autorización del **SO**.

La protección se refiere a un mecanismo para controlar el acceso de los programas, procesos o usuarios, a los recursos definidos por un sistema de computación.

La protección puede mejorar la confiabilidad detectando errores latentes en las interfaces entre los subsistemas. La rápida detección de errores de interfaces puede evitar que un subsistema que no funciona de forma correcta contamine a un subsistema sano.

3.1.7 Redes

Un sistema distribuido es un conjunto de procesadores que no comparten memoria ni reloj, pues cada uno tiene su propia memoria y los procesos se comunican entre sí a través de diversos medios, como canales de alta velocidad o líneas telefónicas.

Los procesadores del sistema se conectan a través de una red de comunicaciones.

Un sistema distribuido permite al usuario el acceso a los diversos recursos que mantiene el sistema.

3.1.8 Sistema intérprete de mandatos

Uno de los programas de sistemas más importantes para un **SO** es el intérprete de comandos, que puede variar en forma y complejidad, pero tiene como función obtener el siguiente enunciado de mandato y ejecutarlo.

Los enunciados de mandato se encargan de la administración de procesos, manejo de E/S, administración del almacenamiento secundario, administración de la memoria principal, acceso al sistema de archivos, protección y redes.

3.2 Servicios del sistema operativo

El **SO** ofrece ciertos servicios para los programas y los usuarios, que varían dependiendo del tipo de **SO**. Estos servicios facilitan la tarea de programar:

☐☐**Ejecución de programas:** El sistema debe ser capaz de cargar en memoria un programa y ejecutarlo.
☐☐**Operaciones de E/S:** Ya que el programa de usuario no puede ejecutar directamente operaciones de E/S, el **SO** debe ofrecer alguna forma de llevarlas a cabo.
☐☐**Comunicaciones:** Hay varias situaciones donde un proceso debe intercambiar información con otro, y lo pueden hacer mediante *memoria compartida* o por *paso de mensajes*.

- **Detección de errores:** El **SO** debe estar constantemente pendiente de errores, que pueden darse en el hardware o en un programa de usuario. Por cada tipo de error, el **SO** debe emprender la acción adecuada, para asegurar un funcionamiento correcto y consistente.

Existe además *otro conjunto de funciones* del **SO**, no para ayudar al usuario, sino para asegurar un funcionamiento eficiente del sistema.

- **Asignación de recursos:** El **SO** administra varios tipos de recursos entre diferentes procesos, usuarios. Para esto cuenta con *rutinas de planificación* (como la CPU) con diferentes criterios.
- **Contabilidad:** Deseamos *llevar un control de cuáles usuarios utilizan cuántos recursos del computador y de qué tipo*. Estas anotaciones pueden tener fines contables o simplemente para recopilar estadística de uso, las cuales pueden ser una valiosa herramienta para los que desean reconfigurar el sistema.
- **Protección:** Cuando varios trabajos se ejecutan concurrentemente, uno, no debería interferir con el otro. La protección implica revisar la validez de todos los parámetros que se pasan en las llamadas al sistema y asegurar que todo el acceso a los recurso del sistema esté controlado. También es importante la seguridad del sistema con respecto a personas ajenas.

3.3 Llamadas al sistema

Las llamadas al sistema proporcionan la interfaz entre el **SO** y un programa en ejecución. Estas llamadas, generalmente, *son instrucciones de lenguaje ensamblador*, aunque otros lenguajes de alto nivel las tienen (C, Bliss, Pascal).

Cada rutina que deba hacer el **SO**, proveniente de un programa se realiza a través de una llamada al mismo (ej.: abrir un archivo).

Las llamadas al sistema pueden hacerse de varias maneras, dependiendo del computador. Frecuentemente se requiere más información que la identidad de la llamada al sistema que se desea.

Para *pasar parámetros* al **SO** se utilizan tres métodos generales. El más sencillo es pasar los parámetros en registros, aunque puede haber más parámetros que registros, pasándose a un registro la dirección de un bloque de memoria con los parámetros faltantes. Otra forma es colocando los parámetros en la pila del sistema.

Las llamadas al sistema se pueden agrupar en cinco grandes categorías:

1. Control de procesos.
2. Manipulación de archivos.
3. Manipulación de dispositivos
4. Mantenimiento de información.
5. Comunicaciones.

3.3.1 Control de procesos y trabajos

Un programa puede terminar normal o anormalmente, dado esto el **SO** debe *transferir el control* al intérprete de mandatos y éste lee entonces el siguiente mandato. En un sistema interactivo, el intérprete de mandatos solo prosigue con el siguiente mandato, pues se supone que el usuario responderá ante cualquier error con el mandato apropiado. En un *batch*, el intérprete termina el trabajo y continúa con el siguiente.

Un proceso o trabajo que ejecute un programa puede desear *cargar* y *ejecutar* otro programa. Ahora ¿a quién le devolvemos el control al terminar este programa?.

Si el control regresa al programa existente, entonces debemos guardar el contenido de la memoria del programa existente, para después recuperarlo. Si ambos programas continúan en forma concurrente, habremos creado un nuevo trabajo o proceso que será multiprogramado. Esto se logra con una llamada al sistema especifica (*crear nuevo proceso*)

Hay otro conjunto de llamadas al sistema útiles para depurar un programa, como llamadas para vuelco de memoria, *rastreo* del programa, que presenta cada instrucción a medida que se ejecuta. Incluso algunos microprocesadores ofrecen un modo *paso a paso*,

donde la CPU *ejecuta una trampa después de cada instrucción*. Esta trampa la captura el *depurador*.

Otros sistemas ofrecen un perfil de ejecución del programa, el cual indica la cantidad de tiempo en que se ejecuta un programa en determinada posición o conjunto de posiciones.

3.3.2 Manipulación de archivos

Para manipular archivos necesitamos hacer llamadas al sistema, por ejemplo para crear o eliminar archivos, debemos mandar la correspondiente llamada con un parámetro (el nombre de archivo). Similar para edición y acceso a directorios, etc.

3.3.3 Administración de dispositivos

Durante su ejecución un programa puede necesitar recursos adicionales. Si los recursos están disponibles se pueden otorgar, y luego devolver el control al programa de usuario, sino deberá esperar. Para solicitar los recursos se deben hacer llamadas al sistema.

3.3.4 Mantenimiento de información

Hay muchas llamadas al sistema con la única finalidad de *transferir información entre el programa usuario y el SO.* Por ejemplo para consultar la hora y la fecha. Otras devuelven: el número de usuarios, versión del **SO**, cantidad de espacio libre en memoria, disco, etc.

El **SO** conserva información de sus trabajos y procesos, y se cuenta con llamadas al sistema para obtener esta información.

3.3.5 Comunicación

Existen dos modelos comunes de comunicación:

Paso de mensajes: la información se intercambia a través de un medio de comunicación entre procesos que ofrece el **SO**. Antes de que la comunicación pueda tener lugar, se debe *abrir una conexión y conocer el nombre del otro comunicador,* ya sea otro proceso en la misma CPU, o de otra. Esto se logra con una llamada al sistema. La comunicación con el *servidor* y los *clientes* se logra a través de llamadas al sistema.

Memoria compartida: los procesos usan llamadas al sistema para *correspondencia de memoria* con el propósito de obtener acceso a las regiones de memoria que pertenecen a esos procesos.

3.4 Programas de sistema

Los programas del sistema ofrecen un entorno más cómodo para el desarrollo y ejecución de programas y pueden dividirse en varias categorías:

- *Manipulación de archivos*: estos programas crean, eliminan, copian, renombran, etc. archivos y directorios.
- *Información de estado*: algunos simplemente solicitan la fecha y la hora, espacio libre de memoria, disco, número de usuarios o información de estado similar.
- *Modificación de archivos*: editores de texto, etc.
- *Apoyo a lenguajes de programación*: compiladores, ensambladores e intérpretes de distintos lenguajes.
- *Carga y ejecución de programas*: el sistema puede ofrecer cargadores absolutos, relocalizables, depuradores, etc.
- *Comunicaciones*: proporcionan mecanismos para crear conexiones virtuales entre procesos, usuarios y diferentes sistemas de computación; permiten a los usuarios enviar a la pantalla de los demás, enviar correo electrónico, etc.
- *Programas de aplicación*: la mayoría de los **SO** se entregan con programas útiles para solucionar problemas comunes (gráficos, juegos, etc.).

Quizás el más importante es el *intérprete de comandos*.

Existen dos maneras de poner en práctica los mandatos al sistema:

1. El *intérprete de comandos* contiene el código para ejecutarlo.
2. Se proporciona un programa especial.

3.5 Estructura de sistemas

Un método común para construir un **SO** es dividirlo las tareas en pequeños fragmentos bien distinguibles.

3.5.1 Estructura sencilla

Algunos sistemas fueron diseñados como pequeños y no se dividieron cuidadosamente en módulos, además estaban limitados por el hardware.

Todo lo que se encuentra por debajo de la interfaz de llamadas al sistema y por encima del hardware físico es el núcleo. Este proporciona mediante llamadas al sistema, el sistema de archivos, la planificación de la CPU, la administración de la memoria y demás.

Las llamadas al sistema definen la *interfaz con el programador* de UNIX.

3.5.2 Enfoque por capas

Con apoyo de hardware, los sistemas operativos se pueden dividir en fragmentos más pequeños. Esto proporciona un mejor control por parte del **SO**, hacia los programas de usuario y el sistema en general.

La modulación de un sistema puede lograrse de varias maneras, pero la más atractiva es el enfoque por *capas*, que consiste en dividir el **SO** en varias capas (niveles), cada una construida sobre las inferiores. La capa más baja es el hardware, la más alta, la interfaz de usuario.

Una capa de **SO** es la implantación de un objeto abstracto que contiene datos y operaciones que pueden manipular esos datos.

La principal ventaja del enfoque por capas es la *modularidad*. Las capas se seleccionan de manera que cada una utilice funciones y servicios únicamente de las capas inferiores. La depuración se realiza por capas, esto simplifica el diseño y la implantación del

La mayor dificultad en el enfoque de capas es definir los niveles.

3.6 Máquinas Virtuales

Algunos sistemas extienden el esquema por capas aún más, permitiendo que los programas de sistema puedan llamarse fácilmente desde las aplicaciones. Como antes, aunque los programas de sistemas se encuentran en un nivel superior al de otras rutinas, las aplicaciones pueden considerarse que se encuentran en un nivel inferior al de otras rutinas. Puede considerarse que todo se encuentra por debajo de ellas en la jerarquía, como si esas rutinas formaran parte de la misma máquina. Este enfoque por capas tiene su condición lógica en el concepto de *máquina virtual*.

Al emplear técnicas de planificación de la CPU y de memoria virtual, un **SO** puede crear la ilusión de procesos múltiples que se ejecutan cada uno en su propio procesador con su propia máquina (virtual). Por supuesto, normalmente el proceso posee requisitos adicionales, como llamadas al sistema y un sistema de archivos que no ofrece el hardware por sí mismo. Por otra parte, el enfoque de máquina virtual no ofrece ninguna función adicional, pero sí una interfaz que es idéntica al *hardware subyacente*. A cada proceso se le otorga una copia (virtual) del computador subyacente.

Los recursos del computador físico se comparten para crear máquina virtuales. Una terminal normal de tiempo compartido ofrece al usuario la función de la consola de la máquina virtual.

Una de las mayores dificultades se refiere a los sistemas de disco. La solución es proporcionar discos virtuales, idénticos en todos los aspectos excepto en el tamaño.

De esta manera se proporciona a cada usuario su propia máquina virtual, y entonces pueden ejecutar cualesquiera de los paquetes de software disponibles en la máquina subyacente.

El concepto es difícil de implantar. Debe tener un modo usuario virtual y un modo monitor virtual, y debe proporcionar un duplicado *exacto* de la máquina subyacente.

Una desventaja obvia es la consumición de recursos por parte de la máquina virtual, y otra son los tiempos de respuesta, que se hacen más lentos.

Dentro de las ventajas se observa que cada máquina virtual está aislada de las demás, por lo que hay protección completa del sistema. Por otra parte, los recursos no se pueden compartir directamente. Hay dos maneras de compartir:

1. *Compartir un minidisco*: sigue el modelo de un disco físico compartido, pero no se pone en práctica mediante software.
2. *Red de máquinas virtuales*: Igual que la física pero virtual.

Este sistema de máquina virtual es perfecto para la investigación y desarrollo de sistemas operativos.

3.7 Diseño e implantación de sistemas

3.7.1 Objetivos del diseño

El primer problema a resolver en el desarrollo de un sistema es el *objetivo* y *sus especificaciones*. En el nivel más alto, el diseño del sistema se verá afectado por la selección del hardware y el tipo de sistema: batch, tiempo compartido, monousuario, multiusuario, distribuido, tiempo real o propósito general. Por otra parte pueden dividirse los requisitos en dos grandes grupos:

- *Objetivos del usuario*: Los usuarios desean ciertas propiedades como: cómodo, fácil de aprender y de usar, confiable, seguro, rápido.
- *Objetivos del sistema:* el **SO** debe ser fácil de diseñar, implantar y mantener, además de ser flexible, libre de errores, etc.

La especificación y el diseño de un **SO** son tareas sumamente creativas. Aunque no haya nada cierto, existen algunas ideas generales:

3.7.2 Mecanismos y políticas

Los mecanismos determinan cómo realizar algo, las políticas deciden qué se hará.

La separación entre estas áreas es importante para lograr flexibilidad. Lo ideal es un mecanismo general, donde un cambio de política, implique solo un cambio de parámetros en el mecanismo.

Las decisiones políticas son importantes para todos los problemas de asignación y de planificación de recursos cuando es necesario decidir si se asigna o no un recurso.

3.7.3 Implantación

Se puede emplear lenguaje de alto nivel para programar un **SO**, conllevando las mismas ventajas que programar cualquier aplicación (es más fácil).

Es probable que las principales mejoras en el rendimiento provenga de mejores estructuras de datos y algoritmos y no de un código más limpio; además, aunque los **SO** sean sistemas muy grandes, sólo una pequeña porción del código es decisiva para conseguir gran productividad: el administrador de memoria y el planificador de la CPU son quizás las rutinas más críticas. Después de escribir el sistema y mientras funciona correctamente, se puede identificar las rutinas que provocan cuellos de botella y sustituirlas con sus equivalencias en lenguaje ensamblador.

Para identificar los cuellos de botellas debemos supervisar el rendimiento del sistema y agregar el código necesario para calcular y presentar las mediciones de su comportamiento.

Otra posibilidad es calcular y mostrar en tiempo real las mediciones del rendimiento.

3.8 Generación del sistema

Lo más habitual es que los **SO** se diseñen para ejecutarse en cualquier máquina, en una variedad de instalaciones y con diversas configuraciones de equipamiento periférico.

Normalmente el **SO** trae un programa de generación del sistema que preguntará: la CPU del sistema, la memoria disponible, los dispositivos, las opciones, etc.

Esta información se puede utilizar para modificar una copia del código fuente del **SO** y luego compilarlo todo, formando un **SO** a medida.

Otra posibilidad es construir un sistema completamente dirigido por tablas. Todo el código siempre formará parte del sistema, y la selección se efectuaría en el momento de la ejecución.

Las principales diferencias entre estos enfoques son el tamaño y la generalidad del sistema obtenido, y la facilidad de modificación al cambiar la configuración del hardware.

4. Aspectos Estructurales de los Sistemas Operativos

4.1 Aspectos Estructurales

4.1.1 Concepto de Sistema

"Un sistema es un conjunto de objetos y un conjunto de relaciones entre esos objetos". Las propiedades de un sistema derivan de los atributos de los objetos que la componen y de la naturaleza de las relaciones entre ellos.

4.1.2 División en tiempos de un sistema

Las sucesivas etapas que experimenta una unidad de trabajo que atraviesa un sistema de computación son:

- **Tiempo de pre-procesamiento**.
  - Tiempo de compilación.
  - Tiempo de combinación.
  - Tiempo de demanda.
  - Tiempo de selección.
  - Tiempo de activación.
- **Tiempo de procesamiento**.
- **Tiempo de post-procesamiento**.

Los eventos específicos asociados con cada tiempo, el número de tiempos que pueden discretizarse y las vistas de estos tiempos definen la flexibilidad del **SO**.

4.1.3 Tiempo de pre-procesamiento

En este tiempo se ejecutan todas las acciones necesarias para preparar un trabajo para ser corrido y al sistema para que pueda correrlo.

4.1.3.1 Tiempo de compilación

Tiempo durante el cual un programa fuente es un dato para un mecanismo, denominado compilador o ensamblador, que lo traduce a programa objeto ejecutable o bien a una forma próxima a la requerida por los mecanismos de procesamiento del sistema. El output del compilador es el input del programa combinador.

4.1.3.2 Tiempo de combinación

Tiempo en el que se realiza la combinación o link-edición, que consiste en enlazar un programa objeto junto con rutinas del sistema y con otros programas de aplicación previamente compilados, produciendo un módulo directamente ejecutable. La combinación puede realizarse después de la compilación o bien en el tiempo de activación.

4.1.3.3 Tiempo de demanda

Tiempo durante el cual un usuario reclama la atención del sistema.

En los primeros sistemas, es el tiempo en el que un lote de tarjetas de control es ingresado al sistema.

Las demandas son monitoreadas por un "receptor de demandas" quien las valida sintácticamente y las graba en una "cola interna de trabajos en espera" (C.T.E). Este mecanismo receptor de demandas generalmente es denominado "Reader" o "Command Interpreter".

4.1.3.4 Tiempo de selección

Tiempo durante el cual el sistema decide posponer u otorgar la atención a la demanda.

En esta etapa, el **SO** elige de la C.T.E el próximo trabajo a ser activado, aplicando algún determinado criterio.

El mecanismo de selección es normalmente un "Scheduler" (comúnmente asociado a procesamiento Batch). Él selecciona una unidad de trabajo (JOB) en función de algún criterio de selección, que pueden ser:

- Punto de vista del usuario:
  - Prioridad relativa.
  - Clases de trabajos.
  - Tiempo límite de iniciación.
  - Tiempo límite de finalización.
  - etc.
- Punto de vista de utilización de recursos:
  - Recurso crítico.
  - Requerimiento de CPU.
  - Necesidad de memoria.
  - Consumo de I/O.
  - Tamaño del programa.

4.1.3.5 Tiempo de activación

Tiempo durante el cual se ejecutan acciones para iniciar un programa. Estas acciones pueden incluir la asignación de dispositivos para las operaciones de E/S, la asignación de conjuntos de datos y la asignación de ubicaciones en memoria. Si estas medidas se posponen, eventualmente llevándolas al tiempo de procesamiento, el sistema deviene más flexible, a costo de una potencial sobrecarga en el ámbito de procesamiento.

Una vez que un trabajo tiene asignado todos aquellos recursos que necesita en forma estática, el mismo se encuentra listo para ser activado; entonces se graba un elemento en la cola de procesos activos (C.P.A.).

Este elemento es creado por el activador para representar al programa en el sistema (se llama Task Control Block (T.C.B.) o Process Control Block (P.C.B.)), y representa sus capacidades, sus deseos y en ciertos casos su historia en el uso del sistema.

4.1.4 Tiempo de procesamiento

Es el período en el cual decimos que un programa está activo.

En los sistemas de serie simples, el tiempo de procesamiento es un intervalo continuo de tiempo durante el cual el programa ejecuta sus operaciones hasta su conclusión.

En sistemas de multiprogramación y de tiempo compartido, el tiempo de procesamiento es el intervalo durante el cual el programa está representado en el sistema por algún elemento (T.C.B. o P.C.B.) que describe el estado y las capacidades del programa. En estos sistemas hay períodos de tiempo entre la activación y la culminación durante los cuales el programa no está utilizando ninguno de los recursos del sistema, o bien no está usando la CPU.

Es durante el tiempo de procesamiento cuando un programa es usuario de la arquitectura ampliada a través de una serie de servicios provistos por componentes del **SO**.

El principal mecanismo asociado a este tiempo es el "Dispatcher", que se encarga de conmutar la CPU.

4.1.5 Tiempo de post-procesamiento

Es el período durante el cual el sistema evacúa un programa ya completado. Los recursos usados por el programa le son retirados y se hacen accesibles.

En ciertos sistemas es el tiempo durante el que el output producido por el programa es impreso o grabado en un dispositivo.

4.1.6 Evolución

Dado que los servicios del tiempo de pre-procesamiento pueden ser implícitamente invocados, ellos pueden ser ejecutados en un número distinto de pasos hasta que el programa ingrese al ámbito de procesamiento.

4.1.6.1 Tiempo Simple

En los primeros sistemas de procesamiento Batch, sólo se tiene el monitor que carga y descarga el único compilador. En ellos se fusionan prácticamente todos los tiempos de pre-procesamiento en el compilador, por eso decimos que eran sistemas de un solo tiempo.

Los servicios del tiempo de demanda son satisfechos por un operador o por un programa monitor residente que llama al compilador o a un programa del usuario según lo especificado en el control-stream.

El compilador realiza todas las asignaciones de las ubicaciones en memoria 8compilación absoluta) y también de los dispositivos que atenderán las E/S, de modo que las direcciones reales de memoria y las direcciones reales de los periféricos son fijadas y compiladas dentro del programa. El resultado de esta compilación puede ser un programa directamente ejecutable.

La activación es simplemente el acto de cargar un programa dentro de la máquina en las direcciones absolutas generadas por el compilador.

Desventajas: inflexibilidad, derivada de la asignación temprana de memoria y dispositivos e imposibilidad de combinar programas escritos en diferentes lenguajes de programación.

4.1.6.2 Combinar y activar

Luego de los sistemas de serie simple nace la idea de distinguir un "tiempo de combinar y activar" o "tiempo de link-edición y carga", como un tiempo definido para demorar la asociación del programa a dispositivos, conjunto de datos y ubicaciones de memoria; y para facilitar la combinación de programas escritos en distintos lenguajes.

4.1.6.3 Carga

Una de las primeras divisiones del único tiempo, derivó en la aparición de un "tiempo de carga", que fusionaba inicialmente los tiempos de selección y activación.

El tiempo de selección es aquel en el cual el monitor identifica una demanda de ejecución, o bien cuando es notificado del final de una ejecución. Se convoca ahora al "Loader" para que cargue el programa objeto en memoria, asignándolo en memoria y los dispositivos y programas precompilados son enlazados (link-editados) al programa a ser cargado. O sea que la operación de carga incluye la asignación de memoria y la carga, junto con el programa, de las rutinas de manejo de dispositivos de E/S.

Característica importante: capacidad del "Loader" de cargar el programa en cualquier conjunto de ubicaciones contiguas de memoria de dimensiones apropiadas. Las rutinas de E/S también deben ser reasignables.

El compilador genera ahora programas reasignables (compilación relativa) y asignación simbólica de periféricos.

En la etapa final de la carga, el "Loader" ubica el programa en memoria principal y le transfiere el control.

El monitor debe incluir un conjunto de tablas representando los recursos disponibles del sistema y un conjunto de bloques de control representando la asociación de nombres simbólicos de archivos con nombres de dispositivos reales que los contienen.

Los bloques de control deberían estar en el ámbito de procesamiento.

4.1.6.4 Link Edición

Un posterior desarrollo a la estructura de pre-procesamiento proviene de la implementación de link-edición como tiempo independiente.

Se logra así ampliar las posibilidades de modo que una serie de programas usuario podían ser combinados independientemente de la asignación y activación.

Ahora sólo el resultado de la combinación o link-edición es sometido al tiempo de activación.

4.1.6.5 Asignadores

Posteriormente, la asignación de conjuntos de datos y dispositivos es separada de la asignación de ubicaciones de memoria y ésta, a su vez, separada de la carga del programa en la memoria principal.

La necesidad de desarrollar una función de asignación nace como consecuencia del incremento del número de recursos del sistema.

Esta función puede ser concebida como una clasificación de los recursos del sistema en clases o tipos y asociar un asignador o "allocator" para cada clase o tipo de recurso. En este caso el tiempo de activación es el tiempo que lleva ejecutar todas las funciones de asignación que requiera un programa.

Cada uno de los asignadores debe tener acceso a tablas que reflejen la disponibilidad de recursos particulares.

Una decisión importante en el diseño de un **SO** es determinar quién y cuándo pueden o deben solicitar los servicios de los distintos asignadores.

Una de las posibilidades de diseño que existen, es realizando todas las asignaciones como requisito previo para la activación (asignación estática).

4.1.6.6 Conclusiones

Dependiendo del tipo de **SO**, de lo sofisticado del mismo y de la capacidad de procesamiento y almacenamiento del equipo, algunos recursos pueden ser asignado y ciertas funciones pueden ser enlazadas o brindadas a un programa en distintos momentos.

Cuanto más se anticipe o retrase la asignación de recursos y/o la combinación de funciones, menos o más flexible, respectivamente, deviene el sistema.

4.1.7 Servicios del sistema y programas de trabajo

El procesador de control-streams invoca funciones que preparan el programa para ser corrido y preparan al sistema para que el programa pueda correr. Por otro lado, los servicios del tiempo de procesamiento (run-time) son invocados a través de macros.

La estricta división que proveen servicios en el tiempo de pre-procesamiento pueden ser organizados en componentes que operan como "programas de trabajo independientes" (característica de compiladores y link-editores, que frecuentemente son excluidos de los **SO**).

4.1.8 Servicios del sistema y privilegios

Si las funciones de demanda, selección y activación son simplemente programas de trabajo independientes, ¿Cómo pueden ser demandadas, seleccionadas, activadas y corridas?.

Una solución es la de un operador que fuerce la selección y activación de los mecanismos de demanda, selección y activación, de modo que éstos estén continuamente representados en el ámbito de procesamiento por un elemento del sistema (TCB). Estos programas estarán inactivos (latentes) hasta que un control-stream arribe al sistema y en ese instante se tornarán listos para correr.

También es común concebir demanda, selección y activación como funciones del **SO** cuyas relaciones con el resto del sistema difieren de las que tiene éste con el compilador y el combinador. Estas relaciones son consecuencia de la necesidad de estos mecanismos de tener acceso a tablas y archivos que no son accesibles a cualquier programa de trabajo.

Estas relaciones especiales se ven reflejadas por ser estos mecanismos los únicos programas del **SO** con acceso a ciertos archivos del sistema. Otra forma de reflejar esta especial relación son garantizar siempre espacio de memoria para que sean corridos, correrlos a niveles especiales de prioridades del sistema, o brindarles mayores privilegios en el ámbito de procesamiento. Estos privilegios ampliados dan la posibilidad de referenciar directamente posiciones de memoria no asignadas a programas de trabajo, acceder a funciones del **SO** sin control previo, o correr en niveles elevados de prioridad.

4.1.9 Subsistemas

Ciertos servicios del tiempo de pre-procesamiento pueden combinarse dentro de subsistemas. Un compilador, por ejemplo, puede tener su propio editor, combinador y cargador.

Un subsistema puede presentar un conjunto mejorado de interfaces del tiempo de procesamiento. Este es un único servicio macro-enriquecidos empaquetados para facilitar la corrida de programas que requieren de ellos sin molestar a aquellos programas que no los necesitan.

4.2 Asignación de recursos

Existen recursos que pueden ser asignados tanto en el tiempo de pre-procesamiento como en el tiempo de procesamiento, lo cual marca la diferencia entre asignación estática y dinámica.

4.2.1 Asignación Estática

Durante el tiempo de pre-procesamiento, luego de haber seleccionado un trabajo, el sistema puede comenzar a acumular algunos recursos para ese trabajo. En los sistemas de multiprogramación, en este tiempo no se asigna tiempo de CPU ni canales, pero si pueden asignarse conjunto de datos, dispositivos y memoria.

El problema principal, para el usuario, es que el trabajo demandado puede experimentar una demora indefinida antes de que sea iniciado dado que todos los recursos deben ser adquiridos conjuntamente.

Para el sistema, está el inconveniente de comprometer un recurso que puede no ser realmente necesitado por el programa, denegando ese recurso a otros usuarios potenciales.

La ventaja para el usuario de tener todos los recurso asignado previo a la ejecución del un trabajo, es que una vez iniciado el mismo, éste asegura los servicios requeridos hasta su finalización.

4.2.2 Asignación Estática

Una solución diametralmente opuesta es que el Scheduler selecciones un trabajo sin asignarle ningún recurso. Los recursos son entonces adquiridos en forma dinámica mediante macros que forman parte del ámbito de procesamiento, luego de la iniciación del programa.

En este caso, el Scheduling consiste en el acto de conceder permiso para competir por recursos.

La ventaja para el usuario es un arranque más rápido del programa y la para el sistema es la recuperación dinámica de recursos no utilizados. Se recarga el ámbito de procesamiento debido a los múltiples actos de asignación y desasignación.

El *objetivo de un sistema* de asignación dinámica es minimizar la diferencia entre uso nominal y uso real de un recurso.

Uso nominal de un recurso: intervalos de tiempo durante los cuales un recurso es asignado y retenido por un programa aunque éste no lo esté utilizando activamente.

Uso real de un recurso: Intervalo de tiempo en que el recurso es usado efectivamente.

4.2.3 Técnicas de Asignación

Las técnicas de asignación son utilizadas para reducir la posibilidad de que los usuarios queden expuestos a demoras indefinidas cuando los recursos son asignados previo a la ejecución del trabajo (sólo asignaciones estáticas).

El *objetivo* de éstas técnicas es acelerar el ingreso de un programa al ámbito de procesamiento.

4.2.3.1 Staging

Consiste en definir "clases de recursos" y asignar cada clase desde una cola de recursos separada. La ventaja del *staging* para el usuario es que facilita el ingreso del programa.

A medida que es adquirida una clase, el programa retiene el recurso y avanza a la siguiente cola. La desventaja para el sistema es que los recursos son retenidos durante períodos de tiempo en que se está esperando por otros recursos.

4.2.3.2 Aging

Es el proceso de ir aumentando la prioridad de un programa con el transcurso del tiempo, o bien en función del número de negativas recibidas ante la solicitud de un recurso.

Puede definirse un umbral de prioridades que cuando es alcanzado impedirá al sistema que siga asignando recursos hasta que el demandante de requerimiento que superó el umbral sea completado. Se define un umbral de prioridades a partir del cual a un determinado programa se le deben asegurar todos los recursos necesarios para comenzar a correr.

4.3 Administración de Recursos

4.3.1 Introducción

El **SO** como administrador de recursos tiene dos aspectos principales:

1. El **SO** provee mecanismos que permiten compartir los recursos de un sistema, protegiéndolos y resolviendo la competencia por su uso.
2. Ciertos recursos son definidos por el **SO** a través del uso del mismo. Estos recursos son la materialización o conceptualización de estructuras abstractas.

El concepto de administración de recursos tiene tres ideas principales:

1. La idea de recurso.
2. La idea de política para alcanzar un objetivo con la administración y la naturaleza de los programas que los demandan.

4.3.2 Recurso

En un principio eran: el procesador, los canales de E/S, los dispositivos y la memoria.

Actualmente, un recurso se ha transformado en una abstracción definida por el **SO** quien otorga una serie de atributos referentes a la forma de acceso al mismo y a su representación física en el sistema.

Un conjunto de recursos del software, sujetos a la administración del **SO**, pueden ser definidos en un sistema de computación como objetos a los cuales se debe controlar el acceso.

La *función* del **SO** es definir una máquina abstracta, compuesta por recursos abstractos que son convenientemente manejados por los procesos para proteger la utilización de los mismos, para asegurar un uso coherente y para imponer políticas de explotación.

4.3.3 Política

El uso de los recursos debe ser fijado por una política, que es la piedra fundamental en la administración de los mismos.

La decisión clave en materia de política de administración es determinar si la maximización del aprovechamiento de un sistema de computación proviene de una rápida respuesta a una demanda de trabajo o de una mayor eficiencia en el uso del recurso (uso interactivo).

Una política puede ser establecida como un conjunto de funciones para lograr determinados objetivos, como por ejemplo:

- ☐☐Minimizar el flow-time promedio.
- ☐☐Minimizar el número de tareas tardías.
- ☐☐Maximizar la utilización del hardware.
- ☐☐Maximizar la utilización del hardware pero con la restricción de que ninguna respuesta sea superior a un cierto punto límite.
- ☐☐Brindar distintos servicios en ciertos porcentajes.

Ciertos **SO** proveen una interface para definir políticas. Los sistemas más flexibles requieren menos planificación off-line.

Un inconveniente es que en las interfaces de los **SO** no existe una clara diferenciación entre definición de política y operación.

En la mayoría de los sistemas la interface operacional permite ejecutar funciones que afectan a la utilización de recursos.

Otro de los problemas relacionados a la interface de política es la necesidad de formular la política de administración en términos de parámetros que sólo pueden ser comprendidos por profesionales.

Es función central de un **SO** actuar como sustituto y/o complemento de quienes dictan las políticas de administración o explotación de un sistema de computación.

El **SO** cuenta para ello con mecanismos que a través de parámetros que se le dan en la definición del sistema, permiten clasificar los trabajos en varios grupos o clases, y declarar la política a seguir por los algoritmos de asignación de recursos en un sistema en que los recursos son compartidos.

En general un **SO** brinda todos o algunos de los siguientes servicios:

- Una interface para administrar las políticas de explotación del sistema.
- Una interface que permite a un profesional en sistemas ajustar la performance del equipo.
- Una interface que posibilita describir la importancia relativa de un trabajo en particular en términos de su pertenencia a un determinado grupo o clase, a un determinado plazo de finalización, o a una prioridad relativa.
- Una interface para describir las características del consumo de recursos de un determinado proceso.

La tarea del **SO** es administrar la carga de trabajos para lograr objetivos preestablecidos. Esto lo logra a través de su capacidad para asignar o negar recursos como tiempo de CPU, memoria, acceso a periféricos y conjunto de datos. Muchos **SO** inspeccionan periódicamente el sistema para determinar si los objetivos están siendo cumplidos; en caso contrario intentan acciones correctivas.

4.3.4 Influencia de la programación

Un programador emplea ciertas técnicas que implican una forma particular de consumo de recursos, o sea que el estilo personal de un programador ocasionará una forma peculiar de consumo de recursos.

El **SO** podrá penalizar o alentar programas con ciertas características, pero el sistema no podrá llegar a la estructura básica de un programa para modificar su performance.

Puede que un "buen programa" represente una dificultad para el sistema cuando se lo incluye en una mezcla de multiprogramación, por ejemplo.

El estilo de programación determina el tamaño de un programa y consecuentemente la cantidad de recursos que éste empleará. El tamaño estimado de los programas puede afectar las estrategias de administración de recursos de varias maneras:

1. Consideremos un conjunto de programas de tamaño reducido. Cada uno de ellos, usaran pocos recursos y tiempo, y serán una pequeña carga para el sistema. La diferencia entre utilización nominal y utilización real de un recurso puede ser pequeña como para que no se necesite incluir un módulo en el sistema para controlar el uso de recursos en un nivel inferior al de asignación.
2. Consideremos ahora que estos programitas han sido link-editados juntos para formar un único programa, de gran dimensión. Los requerimientos de recursos serán superiores. El sistema deberá adoptar, en este caso, una o ambas de las siguientes estrategias:
    a. Otorgar los recursos sólo cuando ellos son específicamente solicitados.
    b. "Administración dinámica de recursos", basa generalmente en mecanismos que responden a razonamientos de tipo heurísticos, monitorea, con un módulo distinto al de asignación, la forma en que el proceso está usando los recursos asignados y consecuentemente se los amplía o restringe,

cediéndolos y recapturándolos como resultado de un patrón de uso de recursos.

Los recursos que pueden ser administrados de esta forma son sólo los reasignables.

Puede extenderse la reasignabilidad a otros recursos, pero ello dependerá del costo relativo de los mecanismos de asignación y desasignación.

Costo relativo: Costo de los mecanismos de asignación versus el costo por la subutilización del equipo.

4.3.5 Fundamentos Económicos

El valor de un sistema de computación reside en el incremento de los beneficios económicos que obtiene una empresa debido a la incorporación de un sistema de computación.

El valor puede derivar de la reducción de los costos de creación, manipulación, clasificación y almacenamiento de datos; del aumento de la productividad de los usuarios y profesionales ligados al sistema.

El costo de computación incluye el costo del hardware, el costo del staff de profesionales derivados del desarrollo de aplicaciones y del mantenimiento de la información y de los programas; y de los costos derivados del efectivo uso del equipo.

Actualmente, el hardware representa un porcentaje cada vez menor del costo total de un sistema de computación, mientras que los costos de trabajo, relativos a la generación del software crecen en gran proporción.

Por ende, la preocupación de maximizar el uso efectivo del hardware está siendo reemplazada por el deseo de obtener del sistema la máxima respuesta posible a las necesidades del usuario y proveer información actualizada y de gran cantidad en el momento que se la necesita.

El valor de un sistema de computación debe ser medido por el valor de la información que él produce.

Por otra parte, se percibe claramente que el costo de una máquina sobrecargada puede ser tanto o más grande que el costo de una máquina subcargada si, debido a la sobrecarga, se desbarata la productividad económica de los usuarios de terminales.

4.3.6 Límites de la administración de recursos

El grado de administración de recursos en un sistema, la intensidad con que es controlada la performance del sistema, la variedad de tiempos disponibles, y la flexibilidad de la asignación y desasignación de recursos están limitados por la naturaleza de la máquina.

La administración de recursos se debe adaptar y limitar a las posibilidades del hardware, de lo contrario, se corre el riesgo de sub-administrar o sobre-administrar el equipo.

Thruput: evolución de trabajos por unidad de tiempo; o sea la cantidad de trabajos que pueden atravesar la máquina en un período de tiempo.

Conclusión: la administración de recursos debe establecer que el consumo de recursos utilizados para ejecutar estas funciones de administración sea tal que el saldo neto de recursos disponibles permitan soportar una mayor carga de trabajos que la que soportaría sin los módulos de administración.

4.4 Función de los Sistemas Operativos

Objetivos principales:

- *Conveniencia*: Hacer al sistema de computación fácilmente utilizable.
- *Eficiencia*: Maximizar el aprovechamiento del sistema de computación, optimizando el uso de los recursos.

El hecho de que una función sea considerada parte o no de un **SO** depende: de la forma de juzgarlo, de las vistas del **SO**, de la historia del sistema y de la frecuencia en el uso de las funciones.

Funciones generalmente atribuidas a un **SO**:

- Proveer un conjunto de aparentes recursos lógicos que sean más fácil de manipular que el hardware que lo soporta.
- Proveer mecanismos de acceso, secuenciación y protección en un medio en el que los recursos son compartidos.

5 Clasificación de los sistemas operativos

Los **SO** difieren entre sí por presunciones que hacen a:

- ☐☐La competencia profesional de los usuarios.
- ☐☐El número de vistas que presentan a los distintos especialistas.
- ☐☐La amplitud y el grado de elaboración de las funciones que proveen.
- ☐☐El grado de soporte de actividades de desarrollo de programas.

Difieren también en su estructura esencial, como ser:

- ☐☐Un sistema organizado para ejecutar un programa según el resultado de un evento externo.
- ☐☐Un sistema organizado para manejar grandes lotes de programas con multiprogramación

Además difieren porque el hardware de base que los soportan son diferentes.

5.1 Clasificación

1. **Hardware subyacente**, distinguiendo entonces grandes máquinas de pequeñas. *Gran Equipo*: las tareas de secuenciamiento y administración de recursos, es de suma importancia. Es deseable que posea mecanismos que permitan al sistema autogestionarse. Es también importante, que soporte diversas formas de acceso y que brinden servicios como administración de bases de datos, subsistemas de consultas, etc. Su **SO** será, por lo general, *multipropósito*. Las tácticas necesarias para proveer una efectiva administración, son también complejas y consumen gran cantidad de tiempo y espacio de almacenamiento. Estos **SO** se caracterizan por tener altos niveles de independencia de periféricos, secuenciamiento dinámico, táctica de asignación de recursos y un importante conjunto de interfaces.
   Máquina Pequeña: hace un uso menos intensivo de sus recursos. Las limitaciones de los **SO** de pequeñas máquinas se reflejan de varias maneras, una de limitar el tipo de acceso que puede hacerse al sistema, permitirá sólo el acceso desde una terminal para el desarrollo de programas y la ejecución de pequeños programas como "esperados". La administración de archivos, sólo se podrá ofrecer de forma sencilla y con limitado acceso de la información.
2. **Método primario de acceso:**
   a. Sistemas Conversacionales.
   b. Sistemas de Tiempo Real.
   c. Sistemas de Procesamiento por Lotes.
3. **Características del usuario:**
   a. Sistemas de Programación.
   b. Computadoras Personales (PC).
   c. Sistemas de Consultas.
4. **Sofisticación de las funciones particulares**.
5. **Manera tradicional**:
   a. Tiempo Real.
   b. Procesamiento en Lotes (Batch).
   c. Serie Simple.
   d. Multiprogramación básica.
   e. Multiprogramación avanzada.
   f. Tiempo Compartido.
   g. Multipropósitos.
6. **Se consideran las relaciones entre el tamaño del hardware, los tipos de acceso y el sistema usado.**

La facilidad de desarrollo de programas es independiente del tamaño de la máquina y del tipo de acceso a la misma. Ciertos tipos de medios de producción parecen requerir acceso "on-line", a pesar de tratarse de pequeño o gran sistema.

5.2 Tiempo Real

Las funciones asignadas al **SO** requerirán que se adapte al (o los) proceso externo que controla respetando estrictamente un marco temporal impuesto por éste última. La administración de todos los recursos del sistema de computación deberá hacerse para proveer en primer lugar las necesidades dictadas por el proceso externo y al cumplimiento en término de las funciones requeridas. Se acepta en este caso, *sacrificar el aprovechamiento óptimo del hardware* en función de satisfacer los requerimientos de la aplicación en el estrecho margen de tiempo disponible.

5.2.1 Clasificación

- **Adquisición de datos**: por medio de sensores adecuados deben "leerse" datos de un proceso externo y almacenarse convenientemente para un uso posterior.
- **Sistemas de control a lazo abierto**: el sistema de computación adquiere los datos del proceso exterior, los procesa convenientemente y los muestra a un operador humano, quien a base la información mostrada toma las decisiones. El operador puede a través del sistema de computación emitir comandos hacia el proceso externo si lo considera necesario.
- **Sistemas de control a lazo cerrado**: el sistema adquiere los datos del proceso externo, evalúa las desviaciones respecto de los objetivos buscados y emite por si mismo los comando correctivos adecuados.

5.2.2 Características

Particularidades:

- La actividad de procesamiento es disparada por eventos aleatorios externos al sistema de computación, por tanto éste debe contar con los elementos necesarios para lograr y mantener una sincronización con un proceso externo.
- Las actividades de procesamiento de dichos eventos, pueden estar compuestas por una o más tareas que deben ser completadas dentro de un estrecho margen de tiempo, congruente con el proceso externo.
- La utilización de hardware es menos importante que la capacidad de respuesta al medio. El sistema de computación estará íntegramente dedicado a las funciones asignadas y por consiguiente se lo configura de manera tal que garantice una respuesta dentro del tiempo prefijado, aún en los momentos de carga máxima.

El intercambio entre el dispositivo y el sistema puede ser iniciado por el sistema que solicita información al dispositivo (*iniciada central*), o por el dispositivo que envía señales con una frecuencia predeterminada o según sus necesidades (*iniciativa periférica*).

Un sistema de tiempo real se caracteriza por una necesidad mínima de intervención humana.

Generalmente se utilizan para supervisar y/o controlar:

- Líneas de montajes.
- Procesos continuos tales como oleoductos.
- Funciones críticas de pacientes.
- Sistemas de luces de tránsito.
- Experimentos de laboratorios.
- Tráfico aéreo.
- Trayectoria de cohetes.
- Etc.

Normalmente es necesario implementar *interfaces especializadas* entre el proceso externo y el sistema.

El proceso externo puede estar cercano al sistema y en ese caso se considera "local", pero si se encuentra en un lugar lejano, se considera "remoto" y se necesita poder transmitir las señales hasta el sistema de computación.

La base del hardware puede ser muy pequeña o muy grande dependiendo de las necesidades de la aplicación. Es posible agregar periféricos para almacenar datos históricos (con fines estadísticos), terminales para consultas y parametrización e impresoras para producir reportes.

Eventualmente, puede realizarse tareas para reducción y análisis de datos (*recolectados en background*) cuando la demanda de tiempo de procesador (*por parte de las tareas de tiempo real en foreground*) lo permitan.

Los atributos de un computador que lo hacen un buen procesador en tiempo real, son:

- El diseño del manejo y análisis de las interrupciones, ya que deben responder rápidamente.
- El diseño de los mecanismos de conmutación de tareas, ya que deben disparar ejecución de procesos muy rápidamente.
- El poder y flexibilidad de sus mecanismos para determinar intervalos de tiempo y la hora real, ya que se necesita muchísima precisión.
- El uso de múltiples procesadores o múltiples sistemas, también es frecuente en sistemas de tiempo real. Esta multiplicidad se usa para lograr mayor confiabilidad (no puede tolerarse que el sistema salga de servicio) y/o para distribuir funciones en un sistema poderoso y complejo.

5.2.3 Software

Características comunes:

- El software posee una estructura de las conocidas como "event driven", es decir que el sistema selecciona los programas a procesar como resultado de una señal externa de determinada característica.
- El corazón de un **SO** de tiempo real, es un módulo llamado "queue manager" que recibe o solicita las señales del proceso externo, las analiza y hace ejecutar un programa que procesa esa señal.
- La administración del procesador (mecanismos de conmutación de la CPU) es estructurada en forma simple.
- Se les permite a los programas correr hasta su finalización o durante lapsos de tiempo prefijado dependiendo del diseño del sistema.
- No se intenta el balance en la utilización de los canales, CPU o memoria, en función de los factores de consumo de tales recursos.
- Normalmente la aplicación tiene el control del equipo, se desprende de la CPU al finalizar su ejecución o al cabo del tiempo prefijado por el "queue manager".
- Frecuentemente, los programas de aplicación están más cercanos a los dispositivos que en los sistemas que no procesan en tiempo real, por lo tanto comparten la responsabilidad de la integridad del sistema y su coordinación.
- La programación y el desarrollo del sistema están separados de la fase operacional del mismo. Es frecuente que el sistema sea desarrollado en un hardware con un **SO** que soporta desarrollo de programas, diferente del hardware en el que va a ejecutarse la aplicación de tiempo real.
- Toda la información está en línea y todos los programas de procesamiento también, en estado de ejecución o muy próximo a ejecutarse. La gran mayoría son residentes en memoria.

Muchos sistemas de tiempo real son desarrollados íntegramente para una aplicación dad; en ellos se planean convenientemente el diseño del hardware de los programas de aplicación y el **SO**. Se descarta la posibilidad de adoptar un **SO** estándar o adaptado. Como todos los programas son conocidos ya que integran el diseño del sistema total, las relaciones entre ellos pueden ser perfectamente predeterminadas; así como también las secuencias de acceso de cada uno de ellos a los dispositivos comunes del equipo y la sucesión de operaciones de E/S. Este hecho permite que todos o al menos la gran mayoría de las actividades de control de los **SO** tradicionales pueden ser simplificadas o eliminadas.

Un **SO** de tiempo real típico para ser usado en una minicomputadora, realiza una rigurosa partición de la memoria en particiones de foreground (utilizadas en las tareas de

mayor prioridad) y de background (utilizadas por las tareas de menor prioridad). Las particiones de foreground son numerosas de modo de poder soportar el concepto de programas procesados organizados en un gran número de tareas.
Tarea: unidad de programa que puede ser cargada en una partición y ejecutada en la misma. Cada tarea es normalmente un subconjunto de una unidad lógica de mayor dimensión.

El **SO** provee mecanismos de activación y sincronización entre las tareas. Estas pueden transmitirse informaciones entre sí en forma de parámetros, esta comunicación se realiza a través de una partición de memoria definida para tal fin. Normalmente un sistema de este tipo provee una biblioteca de rutinas comunes a todas las particiones. Es común encontrar un lenguaje de comandos que brinda cierta flexibilidad operacional a través del cual, es posible asignar periféricos, arrancar y suspender tareas y cambiar prioridades. El background de estos sistemas es utilizado frecuentemente para soportar compiladores, editores y utilitarios orientados al desarrollo de programas.

5.3 Procesamiento Batch

El concepto de procesamiento "batch" alude a la capacidad de un centro de cómputos de aceptar demandas de trabajo no relacionadas entre sí. Esta forma de trabajo se desarrolló a partir de la aparición de los equipos de segunda generación, donde la velocidad de procesamiento es considerablemente incrementada respecto de equipos de la primera generación. Este incremento tiene como consecuencia el hecho que el sistema de montaje de los trabajos por parte del operador es del orden del tiempo de ejecución de los mismos. Surge entonces la preocupación de manejar este tiempo de montaje a los efectos de disminuir el tiempo de máquina ocioso, además también surge el concepto de desarrollos de programas como extensión de las posibilidades que brinda el compilador, este concepto permite la separación de las tareas de programación y operación debido a que permite al programador desarrollar su programa, enviar una representación física del mismo "lote de tarjetas" al centro de cómputos y juntarse con los resultados después de un cierto tiempo (llamado "turn-around-time"(TAT)).

En sus inicios el procesamiento batch perseguía dos objetivos fundamentales:

- ☐☐Aumentar la productividad del equipo, disminuyendo el tiempo de transición entre trabajos.
- ☐☐Aumentar la eficiencia del programador.

Según el grado de elaboración de los mecanismos de ordenamiento de trabajos y de asignación de recursos, los sistemas *batch* se pueden clasificar en:

- ☐☐Serie simple.
- ☐☐Multiprogramación básicas.
- ☐☐Multiprogramación avanzada.

5.4 Serie simple

Conceptos utilizados:

- ☐☐Control stream.
- ☐☐Soporte de E/S.
- ☐☐Reubicación de programas.
- ☐☐Independencia de periféricos.
- ☐☐Encapsulamiento de la máquina por medio de su **SO** y el paquete de software.

Los primeros sistemas procuraron organizar el lenguaje de programación, los soportes de E/S y las rutinas de utilitarios, en un conjunto organizado.

5.4.1 F.M.S. (Fortran Monitor System)

Se desarrolló par el IBM 709. Excepto por su limitación (sólo admite lenguaje Fortran), el sistema mostró características propias de un **SO** de procesamiento batch, permitía realizar una serie de de compilaciones consecutivas como así también una serie de ejecuciones consecutivas.

Aceptaba directivas sofisticadas de depuración. Y permitía realizar vuelcos de memoria. El monitor leía el job stream y producía la carga del compilador cuando era requerido. Durante la fase de ejecución *no existían funciones del monitor residentes* en el equipo, tampoco existía el concepto de programa no linkeditado, de esta manera, un programa en ejecución tenía completo control sobre el hardware, no existía el concepto de estado de control y de operaciones privilegiadas. Las asignaciones de los dispositivos de E/S eran mantenidas en una tabla por el compilador que asociaba los dispositivos lógicos con los físicos, y para cambiar las asignaciones de periféricos era necesario utilizar el editor.

Un aspecto importante fue el concepto de dispositivos comunes para los job stream, tales como *Sysin* y *Sisout* que permitieron reducir el tiempo de preparación de los strams, definiendo el mismo flujo de datos de E/S común para todos los trabajos. Además fue definido el concepto de cintas comunes a todos los trabajos (Sysut-n) simplificando el tiempo de montaje para cada uno.

5.4.2 IBSYS-IBJOB

Fue el más desarrollado de los sistemas de serie simple. Elevaba el número de lenguajes a tres: Fortran, Cobol y MAP e introdujo la idea de *funciones del monitor residentes* para el soporte de E/S.

Su monitor de base (IBSYS) contenía:

- ☐☐Funciones residentes para el soporte de E/S.
- ☐☐Posibilidad de almacenamiento.
- ☐☐Interface para el IBJOB.

El monitor (IBJOB) constaba de:

- ☐☐Intérprete de sentencias de control que permitía seleccionar el compilador.
- ☐☐Cargador responsable de la asignación de dispositivos de programa, de la carga del mismo y de las subrutinas del sistema solicitadas por el programa.
- ☐☐Biblioteca de rutinas de control de operaciones de E/S.

Las tablas de asignación de periféricos que el FMS tenía acceso desde el editor, ahora son manipuladas por el operador mediante tarjetas de control del IBSYS.

No pueden existir programas no linkeditados y no hay necesidad de mecanismos de protección.

5.4.3 Spooling

Con el spooling, obviamente, el TAT aumenta ya que el tiempo original del programa se agrega el tiempo de ejecución de la rutina y el tiempo de grabación en el dispositivo rápido. Por otra parte se aumentará considerablemente el "troughput".

Troughput: número de trabajos que atraviesan el sistema por unidad de tiempo.

Nace así el concepto de multiprogramación.

5.5 Multiprogramación Batch

Tercera generación. Importante la aparición de los *discos rígidos* como medios de almacenamiento ya que presentaron tres posibilidades fundamentales.

- ☐☐Mayor velocidad de dispositivos de almacenamiento masivo.
- ☐☐Capacidad significativa de memoria auxiliar On-line y la necesidad de organizarla y administrarla para lograr una utilización eficiente.
- ☐☐Posibilidad de direccionamiento directo en la memoria auxiliar, brindando nuevas facilidades para las interfaces entre programas y sus operaciones de E/S.

Dentro de las facilidades brindadas por este nuevo hardware se incluyen:

- ☐☐Aparición y/o mejoramiento de los conceptos de interrupción.
- ☐☐Instrucciones privilegiadas.
- ☐☐Mecanismos de protección.

Los sistemas de multiprogramación fueron diferenciados como de multiprogramación básica y avanzada, observando algunas características:

- La sofisticación de los algoritmos de conmutación de la CPU.
- Los algoritmos de administración de la memoria.
- Los tiempos en los cuales un programa puede adquirir o liberar el uso de un determinado recurso.
- El grado en el cual el **SO** participa en las decisiones que hacen a la política administrativa del sistema.

Comparando los sistemas de multiprogramación con los sistemas de serie simple, notamos como importante avance, la existencia de una cola interna de trabajos. La utilización del disco permitió que el conjunto de sentencias de control (job stream) que conforman un trabajo, fuera internalizado de manera que las funciones de ordenamiento pasen a formar parte del **SO**. La cola de trabajos se hace accesible a este ordenador (scheduler) que puede buscar el trabajo en espera que tenga la prioridad más alta o el trabajo que más convenga al sistema en ese momento. Esta cola aumenta considerablemente la participación del sistema en su administración.

5.6 Multiprogramación básica

Características:

- Requieren un importante trabajo de planificación Off-line.
- Poseen administración rígida de la memoria con particiones fijas o redefinidas por el operador y que no se modifican por largos períodos de tiempo.
- Poseen una gestión de conmutación de la CPU con un mínimo esfuerzo para el balance de la utilización de los recursos del sistema.
- Poseen asignación de archivos y periféricos a los programas válidos durante toda la vida activa de los programas (asignación estática).

Las primeras versiones permitían la ejecución concurrente de un predeterminado número fijo de programas que compartían el sistema en un rígido medio de asignación de recursos. La memoria estaba dividida en particiones definidas en el momento de la instalación.

Los compiladores y ensambladores traducían los programas para ser ejecutados en una determinada partición.

Estos sistemas definían múltiples máquinas independientes (una en cada partición) que eran atendidas independientemente, compartiendo periféricos y el uso de la CPU, las colas de trabajos eran creadas para cada partición y las funciones de ordenamiento se corrían en forma independiente de cada partición, las colas estaban armadas por trabajos compilados y ensamblados para esas particiones.

Las particiones reflejaban el tamaño requerido por los trabajos más grandes que se pensaban correr en cada partición.

En realidad, la prepartición tiende a producir una considerable subutilización de la memoria debido a que se generan programas más pequeños que el tamaño de las particiones.

Otro problema de la prepartición es el aumento en los costos de programación y la disminución de la productividad de los programadores, ya que el desarrollo de una aplicación debe encararse teniendo en cuenta el tamaño de la partición en la que se ejecutará.

Esta restricción tiene dos efectos nocivos que pueden presentarse en forma simultánea o por separado:

1. Una distorsión en la estructura del programa ya que la distribución de funciones en el programa no conforma un fin armarlo acorde a la lógica de la aplicación, sino en función del tamaño de la partición.
2. Un incremento en la dificultad de programación que redunda en una mayor subutilización del equipo como consecuencia de una común actitud de los diseñadores de aplicaciones que tienden a minimizar las posibilidades de que la partición asignada sea insuficiente. Otra manera de subutilizar la memoria deriva de la preasignación de programas a determinadas particiones.

El lenguaje de control (JCL) provee un soporte mínimo del concepto de independencia de dispositivos, asociando dispositivos físicos con nombres simbólicos por medio de la sentencia de control "assign".

Como las memorias eran reducidas y costosas, fue preocupación de los diseñadores desarrollar **SO** pequeños y minimizar el tiempo utilizado por las tareas de administración.

Ha medida que las memorias y el hardware se fueron expandiendo se realizaron los siguientes avances:

- ☐☐Posibilidad de redefinir las particiones desde la consola del operador.
- ☐☐Ensamblado y compilación de programas reasignables, permitiendo la ejecución de un programa en cualquier partición de tamaño adecuado, comenzando a aprovechar el sistema como un grupo de recursos dinámicamente asignables.
- ☐☐Asignación simétrica de dispositivos a través de sentencias de control ya sea por clase o por tipo, permitiendo al **SO** asignar los dispositivos recién en el momento de comenzar la ejecución del programa.
- ☐☐Facilidad de organizar los programas en una estructura de tareas (tasks) permitiendo que un programa ejecutado en una determinada partición solicite que comience la ejecución asincrónica de pogramas en la misma partición y provea los mecanismos de sincronización entre ellos (multitasking).
- ☐☐Memoria virtual, asociada a los mecanismos de paginación y segmentación que permitió la construcción de programas con tamaños mayores a la capacidad de la memoria principal.
- ☐☐Desarrollo de mecanismos de soportes avanzados de administración con facilidades pare centralizar el spooling.

5.7 Multiprogramación avanzada

*La principal diferencia entre multiprogramación básica y avanzada, está dado por el grado en que los **SO** administran los recursos del sistema.*

Los de multiprogramación avanzada son sistemas desarrollados como soporte de grandes máquinas a ser usadas en ámbitos donde la carga de trabajo es impredecible y diversa. Se desarrollan como sistemas de propósito general. Proveen grandes posibilidades de administración de recursos, además de un lenguaje de control considerablemente expandido en cuanto a su poder expresivo.

Características comunes:

- ☐☐Cierto grado de administración dinámica de la memoria, incluyendo la posibilidad de brindar a los programas la cantidad de memoria requerida en el momento de comenzar la ejecución, soportando además conceptos de adquisición y liberación dinámica de la memoria.
- ☐☐Cierta posibilidad de compartir códigos (además de los del **SO**), diseñando áreas especializadas para la residencia en memoria de determinadas rutinas de control que serán compartidas por los distintos programas concurrentes.
- ☐☐Considerable elaboración de los sistemas de bibliotecas con mecanismos para la administración, acceso y actualización de archivos On-line.
- ☐☐Desarrolladas estructuras de envíos de trabajos (job-submission), es más elaborada la lectura de las sentencias de control ya que ahora envía a las colas de trabajos (job-queue) un stream control comprimido.
- ☐☐Cierta dependencia de la gestión del operador. Éste tiene capacidad de determinar el nivel de multiprogramación, cambiar prioridades, cancelar trabajos y colocar trabajos en las colas de espera.
- ☐☐Elaboración de los mecanismos de asignación de la CPU, incluyendo la heurística con el objeto de mejorar dinámicamente la respuesta.

5.8 Tiempo compartido

Es aquel sistema cuya intención principal es distribuir la capacidad de procesamiento entre un conjunto de usuarios asociados cada uno a una terminal de E/S.

Los objetivos principales son brindar un servicio efectivo en cuanto a la disponibilidad y una respuesta eficiente a los requerimientos.

A pesar de las similitudes con los sistemas de multiprogramación en lotes y los sistemas de tiempo compartido, como la conmutación de CPU, la diferencia fundamental radica en los objetivos perseguidos. En multiprogramación y batch es máxima utilización del equipo, en tiempo compartido es la pronta respuesta al usuario.

Los sistemas de tiempo compartido son más efectivos cuando cada interacción entre el usuario y el sistema resulta en una pequeña unidad de trabajo que pueda retornar en alguna forma de rápida respuesta a la terminal.

Responden mejor a una carga homogénea en cuanto a las funciones utilizadas; además una población homogénea facilita a los mecanismos de administración al brindar una aparente simultaneidad de servicios.

5.8.1 Desarrollo de programas

Inicialmente, tiempo compartido estaba concebida para brindar por medio de una terminal una estación de trabajo para el desarrollo de programas.

En general un sistema para el desarrollo de programas debe incluir:

- ☐☐Un conjunto de comandos para la activación, testeo y modificación de programas.
- ☐☐Un sistema de administración de archivos con mecanismos que brinden protección, facilidades para compartirlos entre distintos usuarios y/o programas y mantenimiento de una diversidad de objetos (programas, datos, stream de control, etc.)
- ☐☐Uno o más lenguajes de programación.
- ☐☐Una interface para procesamiento que puede superponerse con el lenguaje de comandos.

Un sistema para desarrollo de programas no necesariamente debe estar asociado a un sistema de tiempo compartido.

Entre las posibilidades orientadas por los sistemas más modernos para el desarrollo de programas, podemos enunciar:

- ☐☐Un lenguaje de comandos extendido y adaptado, creando diferencias sintácticas para los usuarios especializados.
- ☐☐La facilidad de llamar para la ejecución a cualquier programa desde la terminal.
- ☐☐El pasaje de argumentos de un comando a otro.
- ☐☐La definición de reglas que permitan a un conjunto de usuarios compartir archivos, controlando el acceso a través de una identificación de grupo.
- ☐☐La definición de operaciones como "no atendidas" para cualquier programa.

5.8.2 Procesamiento de transacciones

Existen **SO** orientados hacia aplicaciones On-line, donde coexisten las características de **SO** de tiempo real y de tiempo compartido. Se asemejan a los de tiempo real por estar desarrollados para atender la ocurrencia de determinados eventos y en la relativa pobreza de los algoritmos de administración de recursos, y a los de tiempo compartido en que cada transacción que ingresa al sistema proviene de una terminal en la que un usuario realiza sus consultas.

El proceso de ejecución es relativamente sencillo y consiste en recibir y analizar un mensaje proveniente de una terminal, transferir dicho mensaje a un programa que lo procesa, algunas referencias a bases de datos en línea para preparar la información necesaria para responder el mensaje y direccionar la respuesta hacia la terminal.

Estos sistemas no son de tiempo compartido en estricto sentido de la palabra. En un sistema de procesamiento de transacciones es común que se permita procesar la transacción hasta su finalización, o por un período de tiempo determinado. Si el procesamiento no se completa en ese período, la transacción se aborta (en lugar de suspenderla).

Características generales que los difieren de los sistemas de multiprogramación en lotes:

- ☐☐Importantes facilidades de telecomunicaciones como parte integrante del **SO**.
- ☐☐Manejo de bases de datos integradas con un sistema sencillo de acceso.

☐☐El sistema de base no provee lenguaje de control ni facilidades para el desarrollo de programas. Aunque, es común que posean importantes herramientas para testeo y determinación de errores de programas en situaciones de simulación.
☐☐El principal constituyente del **SO** es un programa de ejecución cerrada que inspecciona las colas de los programas "listos", "de entrada" y "diferidos". Este programa loop es reactivado cíclicamente conforme al timer de la máquina.
☐☐Poseen un ámbito de procesamiento simple.
☐☐La asignación de memoria es realizada de acuerdo a demandas en bloques de memoria de tamaño fijo.

5.9 Multipropósito

Sistema que posee el atributo de que cualquier método de acceso y uso (subsistemas) son soportados de la misma forma y con igual eficiencia que en el sistema en que fueron desarrollados.

Características:
☐☐Administración dinámica de recursos.
☐☐Soporte de memoria virtual.
☐☐Distintas formas de acceso.

5.10 Procesamiento distribuido

Un sistema constituido por varios procesadores (nodos), es común en estos sistemas.

Poseen especialización de funciones, donde cada nodo posee un conjunto de funciones particulares que combinadas brindan la totalidad del sistema.

Los **SO** de cada nodo, no necesariamente deben ser iguales, incluso el hardware. La comunicación entre nodos puede ser pensada como una responsabilidad de la aplicación, sin embargo, es preferible que esta responsabilidad recaiga sobre los **SO**.

Los conceptos de procesamiento distribuido, se pueden aplicar tanto al procesamiento de transacciones como a los sistemas de procesamientos en lotes.

En los sistemas de procesamiento distribuido se sugiere el hecho de que una aplicación ha sido diseñada para servirse de una familia de procesadores, de manera tal que solo ciertas funciones de la aplicación se ejecutan en cada computadora. La relación entre nodos es más estricta, las interacciones más frecuentes y la dependencia entre los sistemas más estrecha. (ej. Cajero automático).

Hoy en día, la comunicación de sistemas se ha logrado agregando a los **SO** facilidades para que se puedan comunicar, estableciendo "protocolos".

En el desarrollo de estos sistemas, se desea independizar a las aplicaciones de la ubicación física de los datos y de los programas. Para lograr estas transparencias, macroinstrucciones como "OPEN" o "SET" deben poseer la posibilidad de transmitir el requerimiento al nodo que posee los datos necesarios.

5.11 Tipos de acceso

En sistemas donde programación y demanda de servicios se fusionan, ciertos comando aparecen como una extensión del lenguaje de programación y la diferencia entre lenguaje de control y de programación es mínima.

En otros sistemas, el lenguaje está "encapsulado" por el propio lenguaje de comandos.

En general, podemos definir:
En batch: Lenguaje de control = Lenguaje de control.
En On-line: Lenguaje de control = Lenguaje de comandos.

Existen distintos tipos de acceso:

1. Acceso organizado por el operador: este acceso implica la existencia de un "ordenador humano" entre el usuario y el sistema, con períodos fijos de tiempo de uso.
2. Acceso local en modo batch: este acceso eliminó la interface humana, pero incluye un operador del sistema.

3. Acceso remoto en modo batch: este acceso extiende el **SO** al usuario y elimina al operador, la extensión supera las dimensiones físicas del centro de cómputos. Cuando se somete una demanda, un elemento del **SO** se torna activo y lo coloca en una cola de espera.
4. Acceso interactivo en modo batch: este acceso permite al demandante realizar requerimientos dinámicamente. El demandante dispone de lenguaje de manipulación de archivos y bibliotecas, también puede ingresar sentencias de lenguajes de programación y recibir una validación línea por línea de las misma. Cuando un trabajo está listo, envía su control stream al ámbito de procesamiento, a partir de ese momento pierde todo el control sobre el trabajo y solo retorna al contacto con el mismo cuando recibe los resultados.
5. Acceso interactivo para programación: no hay que confundirlo con el anterior, fueron similares en sus orígenes cuando se requería que la totalidad de un programa estuviese grabado para su ejecución (el comando "RUN" implicaba compilación y ejecución). Ahora es posible intercalar actos de programación y ejecución.
6. Acceso a un procesamiento distribuido: el objetivo es permitir a un usuario acceder a procesos o información (datos) de otras computadoras que están interconectadas.

6. Procesos

Un proceso es un programa en ejecución, y en general necesitará ciertos recursos. Estos recursos se asignan al proceso en el momento de crearlo o mientras se está ejecutando.

En la mayoría de los sistemas el proceso es la unidad de trabajo (los sistemas consisten en un conjunto de procesos). En potencia, todos estos procesos pueden ejecutarse concurrentemente.

El sistema operativo es responsable de las siguientes actividades relacionadas con la administración de procesos: Creación y eliminación de procesos, tanto de usuarios como de sistema, planificación de procesos y el suministro de mecanismos de sincronización, comunicación y manejo de bloqueos mutuos entre procesos.

Al conmutar la CPU entre procesos, el sistema operativo puede hacer más productivo el computador.

6.1 Concepto de proceso

Un sistema por lotes ejecuta *trabajos*., uno de tiempo compartido tiene *programas de usuario ó tareas*.

6.1.1 Proceso Secuencial

Informalmente, un *proceso secuencial* es un programa en ejecución. La ejecución de un proceso debe proceder en forma secuencial (en cualquier momento se ejecuta como máximo una instrucción en nombre del proceso).

Un proceso además del código, contiene, por lo general, una *pila* que contiene datos temporales y una *sección de datos* con variables globales.

Un programa es una *entidad pasiva*, mientras que un proceso es una *entidad activa*, con un *contador de programa*.

Aunque dos procesos se puedan asociar a un mismo programa, se consideran como dos secuencias de ejecución separadas; también es habitual que un proceso genere varios procesos más durante su ejecución.

6.1.2 Estado de un proceso

Estados:

- **En ejecución**: Las instrucciones se están ejecutando.
- **En espera**: El proceso está esperando a que ocurra algún suceso.
- **Listo**: El proceso está esperando a que se le asigne a un procesador.

Sólo un proceso puede encontrarse *en ejecución*.

6.1.3 Bloque de control del proceso

En el sistema operativo, cada proceso se representa por medio de su propio *bloque de control del proceso* (PCB o TCB), éste es un bloque o registro de datos que contienen diversa información relacionada con un proceso concreto, incluyendo:

Apuntador.

Estado del proceso.

Contador del programa.

Registros de la CPU: Incluyen acumuladores, registros índices, apuntadores de pila y registros de propósito general, más cualquier información en códigos de condición. Esta información debe guardarse cuando ocurre una interrupción.

Información de planificación de la CPU: incluye una prioridad de los procesos, apuntadores a las colas de planificación y otros parámetros.

Información de administración de memoria: incluye registros límites o tablas de páginas.

Información contable: incluye la cantidad de tiempo real y de la CPU utilizado, límites de tiempo, número de cuenta, números de proceso o trabajo, etc.

Información del estado de la E/S: incluye solicitudes de E/S pendientes, dispositivos de E/S asignados a este proceso, lista de archivos abiertos, etc.

El **PCB** sirve únicamente como depósito de cualquier información que pueda variar de un proceso a otro.

6.2 Procesos concurrentes

Los procesos del sistema pueden ejecutarse concurrentemente. Existen varias razones para esto:

- Compartir recursos físicos.
- Compartir recursos lógicos: varios usuarios pueden interesarse en el mismo elemento de información (por ej. un archivo compartido).
- Acelerar los cálculos: si queremos que una tarea se ejecute con mayor rapidez, debemos dividirla en subtareas, cada una de las cuales se ejecutará en paralelo con las demás (solo si el computador posee múltiples elementos de procesamiento).
- Modularidad: podremos construir el sistema en forma modular, dividiendo las funciones del mismo en procesos separados.
- Comodidad: un usuario puede tener que ejecutar varias tareas a la vez.

La ejecución concurrente que requiere la cooperación entre procesos necesita un mecanismo para la sincronización y comunicación de procesos.

6.2.1 Creación y Terminación de procesos

6.2.1.1 Creación

Durante su ejecución un proceso (*padre*) puede crear varios procesos (*hijos*) nuevos a través de una llamada al sistema para la creación de procesos.

Cuando un proceso crea un subproceso, éste puede obtener sus recursos directamente del sistema operativo o estar restringido a un subconjunto de los recursos del proceso padre, el cual debe dividir sus recursos entre sus hijos, o bien compartir algunos entre varios de sus hijos.

Además de los recursos que un proceso obtiene en el momento de su creación, se pueden pasar datos iniciales del proceso padre al hijo.

Cuando un proceso crea otro nuevo, hay dos alternativas habituales de implantarlo en términos de la posible forma de ejecución:

- El padre continúa su ejecución concurrentemente con sus hijos.
- El padre espera a que todos sus hijos hayan terminado.

6.2.1.2 Terminación

Un proceso termina cuando concluye la ejecución de su último enunciado y solicita al sistema operativo que elimine el proceso. En ese momento, el proceso puede devolver datos a su proceso padre. Un proceso puede provocar la terminación de otro a través de la llamada apropiada al sistema. Por lo general, sólo el padre puede hacerlo. Razones de esto:

- El hijo ha excedido el uso de algún recurso asignado.
- Ya no se requiere la tarea asignada al hijo.

Para determinar el primer caso, el padre debe contar con un mecanismo para inspeccionar el estado de sus hijos. Algunos sistemas no permiten que un hijo exista si su padre ha terminado. En estos sistemas, cuando un proceso termina, entonces también deben terminar sus hijos (*terminación en cascada*, generalmente iniciada por el **SO**).

6.2.2 Relación entre procesos

Los procesos que se ejecutan en el **SO** pueden ser:

Independientes: si no puede afectar o ser afectado por otros procesos que se ejecutan en el sistema. Sus características son:

- Su estado no es compartido de ninguna manera por otro proceso.
- Su ejecución es determinista (su resultado depende exclusivamente de la entrada).

- Su ejecución es predecible (su resultado será el mismo para la misma entrada).
- Su ejecución puede detenerse y ejecutarse sin ocasionar efectos adversos.
- No comparte datos.

Cooperativo: si puede afectar o ser afectado por los demás procesos que se ejecutan. Pueden compartir directamente un espacio lógico de direcciones (usando hilos) o se les puede permitir que compartan datos únicamente a través de archivos. Características:

- Su estado es compartido por otros procesos.
- No puede predecirse el resultado de su ejecución.
- El resultado de su ejecución no determinista.
- Comparte datos.

6.2.3 Hilos

Un "hilo" es una unidad básica de utilización de la CPU y tiene poco estado no compartido. Un grupo de hilos semejantes comparten código, espacio de direcciones y recursos del sistema operativo. El entorno en el cual se ejecuta un hilo se llama *tarea*. Un proceso tradicional equivale a una tarea con un hilo. Una tarea no hace nada si no contiene hilos, y un hilo debe encontrarse exactamente en una tarea. Un hilo posee, por lo menos, su propio estado de registro y por lo general su propia pila. Como el compartimiento es frecuente, el costo de conmutación de la CPU entre hilos del mismo grupo y el de creación de hilos se reducen en comparación con el cambio de contexto entre procesos pesados.

La abstracción que presenta un grupo de procesos ligeros es el de varios hilos de control asociados con varios recursos compartidos. Los hilos pueden ser apoyados por el núcleo., aunque pueden apoyarse por encima del núcleo, a través de un conjunto de llamadas de biblioteca a nivel de usuario.

Mach es *multihilos*, lo cual permite que el núcleo proporcione servicio simultáneamente a varias solicitudes. En este caso los hilos son sincrónicos: un hilo del mismo grupo sólo puede ejecutarse si el hilo en ejecución libera el control. Por supuesto, el hilo en ejecución liberará el control únicamente cuando no esté modificando datos compartidos. En los sistemas donde los hilos son asíncronos, hay que emplear un mecanismo de bloqueo explícito, como en los sistemas donde varios procesos comparten datos.

6.3 Conceptos de planificación

El concepto de multiprogramación es bastante sencillo: un proceso se ejecuta hasta que tenga que esperar, (generalmente por una solicitud de E/S). En un sistema sencillo, la CPU permanecerá inactiva, con la multiprogramación se trata de emplear productivamente este tiempo. Varios procesos se conservan en memoria a la vez, y cuando uno de ellos tiene que esperar, el **SO** le quita la CPU al proceso y se la otorga a otro, y así.

Los beneficios de la multiprogramación son: un aumento en la utilización de la CPU y una mayor *productividad*.

Productividad: cantidad de trabajo desarrollado en un intervalo de tiempo.

6.3.1 Colas de planificación

Conforme los procesos entran en el sistema, se colocan en una *cola de trabajos* formada por todos los procesos que residen en el almacenamiento secundario esperando la asignación de la memoria principal. Los procesos que residen en la memoria principal y que están listos y esperando su ejecución se mantienen en una *cola de procesos listos*, esta lista es generalmente una lista ligada. Un encabezado de la cola de procesos listos contendrá apuntadores al primer y último PCB de la lista. Cada PCB tiene un campo apuntador que indica el siguiente proceso en la cola de procesos listos.

La lista de procesos que espera a un dispositivo de E/S determinado se denomina *cola del dispositivo*, y cada dispositivo tiene su propia cola. Si se trata de un dispositivo

dedicado, la cola nunca tendrá más de un proceso, si es compartido, pueden haber varios procesos.

Un proceso nuevo se coloca inicialmente en la cola de procesos listos. Una vez que se asigna la CPU al proceso y se ejecuta, puede ocurrir uno de estos sucesos:

El proceso puede emitir una solicitud de E/S y colocarse en una cola de dispositivo.

El proceso puede crear un nuevo proceso y esperar que éste termine.

El proceso podría ser extraído de la CPU por la fuerza, mediante una interrupción, y colocarse de nuevo en la cola de procesos listos.

En los dos primeros casos, el proceso cambia eventualmente del estado de espera al estado de listo, y se coloca de nuevo en la cola de procesos listos.

6.3.2 Planificadores

Un proceso transita entre las distintas colas de planificación, y el **SO** de alguna manera debe seleccionar procesos de estas colas. Esta actividad la realiza el *planificador* correspondiente.

6.3.2.1 Planificador a Plazo (OAN – Scheduler)

Selecciona una unidad de trabajo desde una cola de trabajos en espera para pasarlos a otra cola de procesos activos o iniciados, según sea un sistema de dos o tres niveles, desde donde competirán por la CPU. Criterios desde el punto de vista de:

- ☐☐Servicios al usuario: prioridad relativa, clases de trabajos, tiempo límite de iniciación, tiempo límite de finalización.
- ☐☐Utilización de recursos: recurso crítico, requerimiento de la CPU, necesidad de memoria, consumo de E/S, tamaño del programa.

En un sistema por lotes, con frecuencia se presentan más procesos que los que se pueden ejecutar de inmediato; estos procesos se envían a un *spooler*, donde se conservan para su posterior ejecución. El *planificador* a largo plazo (*Scheduler*) selecciona procesos de este depósito y los carga en memoria para su ejecución.

El *Scheduler* se ejecuta con menos frecuencia que el planificador a corto plazo. El *Scheduler* controla el grado de *multiprogramación* (el número de procesos en memoria). Si la tasa promedio de creación de procesos, es igual a la de finalización de procesos, se invoca al *Scheduler* cuando finaliza un proceso. Debido al mayor intervalo de tiempo entre ejecuciones, el planificador a largo plazo puede emplear más tiempo para decidir qué proceso se debe seleccionar para su ejecución. También puede ser importante que la selección sea cuidadosa.

La mayoría de los procesos pueden describirse como limitados por la CPU o limitados por E/S, por eso es importante que el planificador a largo plazo seleccione una buena *mezcla de procesos* limitados por la CPU y limitados por E/S.

En sistemas de tiempo compartido, generalmente no hay *Scheduler*.

El planificador a largo plazo es un evento único en la vida de un programa.

6.3.2.2 Planificador a Corto Plazo

El *planificador a corto plazo* (OBN – *Dispatcher*) selecciona, desde la cola de procesos activos, uno de los procesos listos para ejecución y le asigna la CPU.

La principal diferencia entre el *Scheduler* y el *Dispatcher* es la frecuencia de su ejecución.

Por lo general, este planificador de la CPU se ejecuta una vez cada 10 milisegundos.

6.3.2.3 Planificador a Mediano Plazo

Algunos sistemas operativos, como los de tiempo compartido o multiprogramación avanzada o multipropósitos, pueden presentar un nivel intermedio adicional de planificación, llamado de *mediano plazo* (PMP – ONI, etc).

La idea clave del planificador a mediano plazo es que en ocasiones puede ser ventajoso eliminar procesos de la memoria y de este modo reducir el grado de multiprogramación. Más tarde el proceso se volverá a introducir en la memoria y continuará su ejecución a partir del punto donde se quedó. A este esquema comúnmente se le denomina intercambio (*swapping*). Los intercambios pueden ser necesarios para

mejorar la mezcla de procesos, o porque un cambio en los requisitos de memoria ha comprometido en exceso la memoria disponible, lo que requiere que se libere memoria.

Este planificador tiene por objetivo flexibilizar el sistema sin sobrecargar al planificador a corto plazo quien debe trabajar con especial rapidez conmutando la CPU entre los distintos programas que conforman la mezcla de multiprogramación.

Los procesos que residen en la cola de procesos iniciados (CPI) esperan a que el planificador a mediano plazo (ONI) les permita pasar a competir por el uso de la CPU. El ONI elegirá un programa de la cola de procesos iniciados (CPI) para ser pasado a la cola de procesos activos (CPA) en base a dos grupos de información que recibe:

- Información sobre la utilización que se está haciendo del sistema, como ser: ratio de uso de CPU, ratio de uso de los canales, llenado de memoria, etc.
- Información sobre el grado de progreso de los programas.

El ONI tendrá información permanentemente actualizada del comportamiento de la totalidad de los programas.

Con el advenimiento de sistemas que devienen más flexibles y adaptables a un universo heterogéneo de tareas, un sistema de dos niveles se torna inadecuado por las siguientes razones:

- Complican la decisión del planificador a corto plazo.
- Imposibilidad de considerar la heurística
- Cuando se pretende balancear dinámicamente la mezcla, computamos parámetros tales como "grado de avance de un programa", o "tiempo límite de finalización" se requiere correr un número de funciones de análisis del estado del sistema que cargaría en demasía a la etapa del OBN.

La idea de un tercer nivel es formar una cola de tareas que proviene del planificador a largo plazo o de una partición de tiempo compartido, pero que no son sometidas al OBN hasta que el ONI defina una lista para el mismo.

El ONI podrá operar sobre la base de un tiempo fijo o bien en respuesta a un evento como puede considerarse la detección de una sobrecarga en un momento dado de un determinado recurso, por ejemplo:

- Cuando en un sistema que soporta paginación el número de páginas disponibles en memoria cae por debajo de un cierto nivel.
- Para definir la mezcla de programas activos tendrá en cuenta que el conjunto de páginas de trabajo (CPT) de los programas con los que conforma la mezcla puedan acomodarse sin problemas en la memoria principal.
- Cuando se quiere balancear dinámicamente la mezcla, el ONI reemplaza por ejemplo programas cuyo recurso límite es la CPU por otros cuyos recursos críticos son los canales, para esto recibe información de los administradores de carga, CPU y canales sobre el avance de un determinado programa y de los ratios de utilización de CPU y canales.
- Para un programa con tiempo límite de finalización, se calcula en el momento de su iniciación el monto total de recursos que necesita, expresado en "unidades de servicio", y al final de cada período se controla el grado de avance del programa; y según esté retrasado o adelantado se le modificará, si es necesario, la prioridad relativa.

Unidad de servicio: concepto que permite medir el grado de avance de un programa en una mezcla, conforman una misma unidad los montos necesarios de: ciclos de CPU, demanda de E/S y montos de memoria, permitiendo lograr una medida comparativa de la carga total que representa ese programa en una mezcla.

6.4 Planificación de la CPU

La *planificación* es una función fundamental del sistema operativo.

6.4.1 Ciclo de ráfagas de CPU y E/S

La ejecución de un proceso consiste en un *ciclo* de ejecución de ráfagas de CPU y ráfagas de E/S.

La curva de ráfagas de CPU se caracterizan como exponencial o hiperexponencial. Hay un gran número de ráfagas de CPU de corta duración y un pequeño número de ráfagas de larga duración.

Un programa limitado por E/S, es aquel con muchas ráfagas de CPU breves.

6.4.2 Planificador de la CPU

Siempre que la CPU queda inactiva, el sistema operativo debe seleccionar para su ejecución uno de los procesos de la cola de procesos. El proceso de selección es realizado por el planificador de corto plazo.

Los registros de las colas suelen ser los PCB de los procesos.

6.4.3 Estructura de planificación

Las decisiones de planificación de la CPU pueden efectuarse en una de las cuatro circunstancias siguientes:

1. Cuando un proceso cambia del estado de ejecución a estado de espera.
2. Cuando un proceso cambia del estado de ejecución al estado listo.
3. Cuando un proceso cambia del estado de espera al estado listo.
4. Cuando termina un proceso.

Cuando la planificación tiene lugar únicamente en las situaciones 1 y 4, decimos que el esquema de planificación es *no apropiativo* (una vez que la CPU se ha asignado a un proceso, éste la libera, solamente al terminar o al cambiar al estado de espera); de lo contrario decimos que es *apropiativo*.

6.4.4 Cambio de contexto

Para cambiar la CPU a otro proceso se requiere guardar el estado del proceso anterior y cargar el estado guardado para el nuevo proceso. Esta tarea se conoce como *cambio de contexto*. El tiempo de cambio de contexto es un puro gasto adicional, y varia de una máquina a otra.

Cuanto más complejo sea el sistema operativo, más trabajo hay que realizar durante un cambio de contexto. Las técnicas avanzadas de administración de memoria pueden requerir que con cada contexto se cambien otros datos adicionales.

6.4.5 Despachador

El despachador es el módulo que realmente entrega el control de la CPU al proceso seleccionado por el planificador a corto plazo, implicando:

- ☐☐Cambiar de contexto.
- ☐☐Cambiar a modo usuario.
- ☐☐Saltar a la posición adecuada del programa del usuario para reiniciar el programa.

6.5 Algoritmos de planificación

Para comparar los algoritmos de planificación de la CPU se han propuesto varios criterios que incluyen:

Utilización de la CPU: Su utilización puede variar del 0 al 100% y en un sistema real debe fluctuar entre el 40 y un 90%.

Productividad (*Throughput*): número de procesos que se completan por unidad de tiempo.

Tiempo de retorno (*Turnaround time*): Desde el punto de vista de un proceso en particular, el criterio más importante es cuánto tiempo tarda en ejecutarse ese proceso. El tiempo de retorno, es el que tarda desde que es demandado hasta que finaliza.

Tiempo de espera (*Waiting time*): El algoritmo para la planificación de la CPU no afecta realmente a la cantidad de tiempo durante el cual el proceso se ejecuta o lleva a cabo E/S. EL algoritmo afecta únicamente a la cantidad de tiempo que el proceso espera en la cola de procesos listos.

Tiempo de respuesta (*Response time*): Tiempo transcurrido desde la presentación de una solicitud hasta que se produce la primera respuesta. En un sistema interactivo es posible que el tiempo de retorno no sea el mejor criterio. Con frecuencia un proceso puede producir alguna salida en los primeros instantes y continuar calculando nuevos resultados mientras se presentan al usuario los resultados anteriores.

En la mayoría de los casos se optimiza el promedio, pero en ocasiones puede ser deseable optimizar los valores mínimos o máximos.

6.5.1 Planificación "Servicio por orden de llegada" (FCFS)

Algoritmo más sencillo *First-Come, First Served*. El proceso que primero solicita la CPU es el primero al que se le asigna. Cuando un proceso entra en la cola de procesos listos, su PCB se enlaza al final de la cola

El FCFS el tiempo promedio de espera es bastante largo.

Si hay un proceso de gran tamaño, los demás procesos deben esperar su terminación y esto crea lo que se llama *efecto convoy*.

Este algoritmo es **no apropiativo** y especialmente problemático en tiempo compartido.

6.5.2 Planificación "Primero el trabajo más breve" (SJF)

Este algoritmo asocia a cada proceso la longitud de su siguiente ráfaga de CPU. Cuando la CPU está disponible, se le asigna al proceso que tiene la ráfaga siguiente de CPU menor. Si dos procesos tienen la misma longitud para la siguiente ráfaga de CPU, se utiliza planificación FCFS.

Este algoritmo es *óptimo* ya que ofrece el mínimo tiempo promedio de espera para un conjunto de procesos dado.

El problema real de este algoritmo es conocer la longitud de la siguiente solicitud de CPU. Para la planificación a largo plazo (uso frecuente de este algoritmo) en un sistema por lotes, podemos utilizar el tiempo límite del proceso.

No puede implantarse a nivel de corto plazo de CPU. No hay manera de conocer la longitud de la siguiente ráfaga de CPU. Aunque *no conocemos* la longitud de la siguiente ráfaga de CPU, podemos *predecir* su valor, esperando que sea de longitud similar a las anteriores.

El algoritmo SJF puede ser **apropiativo** o **no apropiativo**. Cuando un nuevo proceso llega a la cola de procesos listos mientras se está ejecutando otro proceso. El nuevo proceso puede tener una ráfaga de CPU menor que lo que resta del proceso que se ejecuta en ese momento.

6.5.3 Planificación por prioridades

El algoritmo SJF es un caso especial del algoritmo general para la planificación *por prioridades*. Se asocia una prioridad a cada proceso y la CPU se asigna al de mayor prioridad. Los procesos con igual prioridad se planifican en orden FCFS.

No existe conversión acerca de los números usados para las prioridades.

Las prioridades pueden definirse interna (internas del **SO**, en relación con: límites de tiempo, requisitos de memoria, el número de archivos abiertos y la tasa de intervalos entre ráfagas de E/S y de CPU) o externamente (externas al **SO**, en relación con: importancia del proceso, tipo y cantidad de fondos que se pagan por utilizar el computador, generalmente políticas).

La planificación por prioridades puede ser **apropiativa** o **no apropiativa**. Cuando un proceso llega a la cola de procesos listos, su prioridad se compara con la del proceso en ejecución.

Un serio problema de los algoritmos por prioridades es el *bloqueo indefinido* o *inanición*. Un algoritmo para planificación por prioridades puede dejar a un proceso de baja prioridad esperando indefinidamente a la CPU.

Una solución es utilizar *envejecimiento* (*aging*), técnica por la cual aumenta gradualmente la prioridad de los procesos que esperan durante mucho tiempo en el sistema.

6.5.4 Planificación Circular (Round Robin)

Diseñado especialmente para tiempo compartido. Se define una pequeña unidad de tiempo (*quantum*), que generalmente varía entre 10 y 100 milisegundos. La cola de procesos listos se trata como una cola circular, el planificador de la CPU la recorre asignando la CPU a cada proceso por un intervalo de hasta un quantum.

Para poner en práctica la planificación RR, mantenemos la cola de procesos listos como una cola FIFO. Los nuevos procesos se agregan al final de la cola de procesos listos. El planificador de la CPU toma el primer proceso de la cola, programa un *timer* para que interrumpa después de un quantum de tiempo y despacha el proceso.

Entonces sucederá una de estas dos cosas: el proceso puede tener una ráfaga de CPU menor que un quantum de tiempo, en cuyo caso el proceso liberará voluntariamente la CPU y el planificador continuará con el siguiente proceso de la cola de procesos listos. Por otra parte si la ráfaga de la CPU del proceso en ejecución es mayor que 1 quantum de tiempo, el timer se activará y provocará una interrupción para el sistema operativo. Se ejecutará un cambio de contexto y el proceso se colocará al final de la cola de procesos listos. El planificador seleccionará el siguiente proceso de la cola.

El tiempo de espera promedio es bastante grande en la política RR.

En el algoritmo de planificación RR, la CPU no se asigna a ningún proceso por más de un quantum de tiempo. Este algoritmo es apropiativo.

El rendimiento de RR depende en gran medida del tamaño de quantum de tiempo, si es muy grande (infinito), la política RR es la misma que la de FCFS. Si es muy pequeño, el enfoque RR se llama *compartir el procesador* y para los usuarios parece (en teoría) que cada uno de los *n* procesos tiene un procesador propio que se ejecuta a 1/*n* de la velocidad real. Sin embargo, hay que considerar el efecto del cambio de contexto en el rendimiento de la planificación RR, por lo tanto el quantum del tiempo debe ser grande respecto al tiempo de cambio de contexto.

El tiempo de retorno también depende del tamaño del quantum. El tiempo de retorno, no necesariamente mejora al aumentar el quantum. En general, puede aumentar si la mayoría de los procesos terminan su siguiente ráfaga de CPU en un solo quantum.

Una regla empírica es que el 80% de las ráfagas de CPU deben ser menores que el quantum de tiempo.

6.5.5 Planificación de cola de múltiples niveles

Una división habitual consiste en diferenciar los procesos de primer plano o *foreground* de los procesos de segundo plano o *background*. Estos dos tipos de procesos tienen requisitos de tiempo de respuesta bastante diferentes por lo que pueden presentar distintas necesidades de planificación. Además, los procesos *foreground* pueden tener una prioridad superior (definida exactamente) a la de los procesos background.

Un *algoritmo de planificación de colas de múltiples niveles* divide la cola de procesos listos en diversas colas. Los procesos se asignan en forma permanente a una cola, generalmente a partir de alguna propiedad del proceso como puede ser el tamaño de la memoria o el tipo de proceso. Cada cola tiene su propio algoritmo de planificación.

Debe existir además una planificación entre las colas, la cual generalmente es una planificación apropiativa de prioridad fija.

Otra posibilidad es usar una porción de tiempo para las colas.

6.5.6 Planificación de colas de múltiples niveles con realimentación

La *planificación de colas de múltiples niveles con realimentación* permite a un proceso moverse de una cola a otra. La idea es separar los procesos con diferentes características en cuanto a ráfagas de la CPU. Si un proceso utiliza demasiado tiempo de la CPU, se pasará a una cola de menor prioridad. Este esquema deja a los procesos limitados por E/S y a los procesos interactivos en las colas de mayor prioridad. Si un proceso espera demasiado tiempo en una cola de menor prioridad se puede mover a una

de mayor prioridad. Esta es una forma de envejecimiento que evitaría el bloqueo indefinido.

Por lo general este tipo de colas se define con los siguientes parámetros:

- El número de colas.
- El algoritmo de planificación de cada cola.
- El método utilizado para determinar cuándo promover un proceso a una cola de mayor prioridad.
- El método utilizado para determinar cuándo degradar un proceso a una cola de menor prioridad.
- El método utilizado para determinar cuál cola entrará un proceso cuando necesite servicio.

La definición de un planificador de colas múltiples niveles con realimentación lo convierte en el algoritmo de planificación de la CPU más general, pero también el más complejo.

6.5.6.1 Aplicación de colas cíclicas realimentadas

Estructura de cola en la cual la cola del dispatcher aparece particionada en *n* subcolas. Se tiene una ley de residencia de los programas que residen en cada subcola, y un conjunto de relaciones entre las mismas.

Una estructura de subcolas permite clasificar dinámicamente programas, según características intrínsecas de los mismos, y tratarlos en forma diferencial.

Para cada subcola se define:

- N=*Iteraciones*: Número máximo de veces que un programa puede recibir la CPU desde una determinada subcola.
- Q=*Quantum*: Período máximo de tiempo continuo que un programa proveniente de esa subcola puede usar la CPU.
- RL=*Regla de residencia*: determina, la permanencia de un programa en una determinada subcola, o su pasaje a otra.
- VL=*Regla de visita*: Controla el servicio de la CPU a cada una de las subcolas, regulando la importancia (P) que se le dará a c/u.

Mediante la aplicación de la regla de residencia se logra clasificar programas según su comportamiento y alojarlos en distintas colas.

El reconocimiento dinámico de programas con distintas características es un ejemplo de "administración heurística de recursos". El principio básico subyacente en el pasaje de un programa de una subcola a otra, es asumir que un programa que ha consumido muchos servicios del sistema en el pasado, tiene aún muchos servicios por consumir.

Dado que lo que se pretende es penalizar a determinados programas y beneficiar a otros, es que se definen distintas "reglas de visita" a cada subcola, cada una de ellas con distinto grado de severidad.

La más severa dice: "La primer subcola recibirá la CPU siempre que tenga un programa en condiciones de servirse de la misma".

Otra regla habitual es: en máquinas que tienen la memoria particionada en foreground y background, es que la CPU estará sirviendo a la cola de bloques representativo de los programas que están en background, siempre que no se solicite la CPU desde un programa residente en foreground.

Otras reglas de visita consisten en definir distintos coeficientes de utilización de servicios para cada una de las subcolas.

La aplicación de una estructura de subcolas permite alcanzar objetivos diametralmente opuestos en sistemas de tiempo compartido y Batch.

Para comprender cabalmente el funcionamiento, los fundamentos y los alcances de este tipo de estructuras se debe tener presente, para cada uno de los **SO** cuales son: los objetivos, la mezcla ideal de trabajos, la naturaleza de los programas, las características de los usuarios, el tipo de información que producen y fundamentalmente cual es el recurso crítico.

6.5.6.2 Aplicación a sistemas de tiempo compartido

El objetivo es minimizar el tiempo de respuesta a los usuarios que interactúan con el sistema a través de una terminal. Se trata de privilegiar a aquellos programas típicamente conversacionales.

Una estructura de subcolas permite, en este tipo de sistemas, distinguir los programas largos de los cortos, para beneficiar a éstos últimos.

La regla de residencia, en este caso, está definida por las "N".

6.5.6.3 Aplicación a sistemas de procesamiento Batch

Es requisito maximizar la utilización de los componentes de E/S, por dos motivos fundamentales:

1. Por el costo relativo de dichos componentes.
2. Porque esta política de maximización contribuye al balance del sistema.

La regla de residencia enunciada como "el más corto primero", que consiste en clasificar los programas según su consumo de E/S y otorgarle la CPU al programa que correrá el menor tiempo antes de solicitar una operación de E/S.

En este tipo de sistemas, la estrategia asignación dispatching deberá ser más compleja y deberá incluir el cálculo y la evaluación de ratio "tiempo de canal / tiempo de CPU". Supongamos que en un momento dado el sistema detecta que la mezcla proporciona una "baja" utilización de la CPU, la solución sería agregar a la mezcla un programa cuyo recurso crítico sea la CPU.

El resultado de introducir este programa en la mezcla puede ser contrario al esperado, el ratio de la CPU puede bajar aún más pues el tiempo ocioso de CPU se debía a que el dispatcher no encontraba trabajos listos para correr debido al tiempo que transcurre entre la demanda de una E/S y su efectiva cumplimentación ya que este tiempo es una función de las colas de espera y del tiempo de búsqueda de registros en los periféricos, entonces el programa agregando a la mezcla aumenta ligeramente la carga de E/S, y el programa que inicialmente se pensó como limitado por la CPU ya no lo es más y además su tiempo de espera para las E/S es el propio más el de todos los programas anteriores, por lo tanto se incrementa el tiempo de espera para los otros programas y el tiempo que transcurre para que un programa esté listo aumenta y por lo tanto baja el ratio de utilización de la CPU en vez de aumentar.

La utilización de subcolas tiende a facilitar el logro del equilibrio entre el dispatching y asignación, asegurando que el programa que solicita la operación de E/S más rápido tenga el control de la CPU siendo la regla de residencia en una subcola para este caso: el tiempo máximo que puede correr un programa antes de solicitar una operación de E/S.

Si el programa efectivamente solicita una operación de E/S antes de ese tiempo máximo que fija la regla de residencia de esa subcola, el programa queda en esa subcola, sino es pasado a la siguiente subcola.

Esta regla asume el concepto de la regla heurística, según la cual: un programa que ha demandado gran cantidad de recursos de canal, lo seguirá haciendo en el futuro.

Una variante a los expuestos sería tomar como parámetro para definir la regla de residencia de los programas de las subcolas, no la frecuencia de utilización de canales, sino el tiempo de espera de recursos de CPU.

6.5.6.4 Prioridades

Un aumento en la prioridad de un programa representa una menor utilización de los canales. Si un programa de prioridad superior tendrá el uso de la CPU por un lapso extenso para correr sin ser penalizado por no requerir una operación de E/S, por lo tanto se disminuye el uso de los canales de E/S.

6.6 Planificación de procesadores múltiples

Si hay múltiples CPU, el problema de planificación se vuelve más complejo.

Uno de los factores principales es el tipo de procesadores que entran en juego, los cuales pueden ser idénticos (sistema *homogéneo*) o distintos (sistema *heterogéneo*). Si los procesadores son diferentes, las opciones son relativamente limitadas. Cada

procesador tiene su propia cola y su propio algoritmo de planificación. Los procesos están tipificados intrínsecamente por su estructura, y deben ejecutarse en un procesador determinado.

Si hay varios procesadores idénticos, pueden *compartir cargas*. Sería posible proporcionar una cola distinta a cada procesador, pero en esta situación un procesador podría estar inactivo. Para evitarlo utilizamos una cola común de procesos listos; todos los procesos entran en esta cola y se planifican en cualquier procesador disponible.

Con este esquema puede emplearse una de dos estrategias de planificación. En una de ellas, cada procesador se planifica a sí mismo. Cada procesador examina la cola común de procesos listos y selecciona un proceso para ejecución. Si tenemos varios procesadores que tratan de acceder a una estructura de datos común y actualizada, hay que programar con cuidado cada procesador. Debemos asegurar que dos procesadores no elijan el mismo proceso, y que no se pierdan procesos de la cola. La otra estrategia evita este problema estableciendo un procesador como planificador para los demás, creando así una estructura amo-esclavo, esto es, el *multiprocesamiento asimétrico*.

6.7.2 Simulaciones

Para conseguir una evaluación más precisa de los algoritmos de planificación, podemos usar *simulaciones*, las cuales implican programar un modelo del sistema de computación. Las estructuras de datos de software representan los principales componentes del sistema. El simulador tiene una variable que representa el reloj y, al aumentar el valor de esta variable, el simulador modifica el estado del sistema para reflejar las actividades de los dispositivos, los procesos y el planificador. Al ejecutar la simulación, se recopilan e imprimen estadísticas que indican el rendimiento del algoritmo.

Los datos para dirigir la simulación pueden generarse de varias maneras. El método más común utiliza un generador de números aleatorios, programado para generar procesos, tiempos de ráfaga de CPU, llegadas, salidas, etc, de acuerdo con distribuciones probabilísticas. Si la distribución se define empíricamente, se toman mediciones del sistema en estudio; los resultados se emplean para definir la distribución de sucesos en el sistema real y esta distribución puede usarse para dirigir la simulación.

Sin embargo, una simulación dirigida por distribuciones puede ser inexacta, debido a las relaciones entre sucesos secuenciales en el sistema real. La distribución de frecuencias sólo indica cuántos sucesos de cada tipo ocurren; no puede señalar nada acerca del orden. Para corregir este problema utilizamos *cintas de rastreo* que se crean controlando el sistema real y registrando la secuencia de sucesos. Luego se utiliza esta secuencia para manejar la simulación. Las cintas de rastreo son una forma excelente de comparar dos algoritmos con el mismo conjunto de entradas reales. Este método puede producir resultados muy precisos para sus entradas.

No obstante, las simulaciones pueden ser muy costosas y con frecuencia requieren horas de uso del computador. Por último, el diseño, codificación y depuración del simulador pueden ser una tarea de gran magnitud.

6.7.3 Implantación

La única manera completamente exacta de evaluar un algoritmo de planificación es codificarlo y colocarlo en el sistema operativo para ver cómo funciona.

El mayor problema de esta estrategia es su costo. Se incurre en gastos no sólo para codificar el algoritmo y modificar el sistema operativo para que lo apoye junto con sus estructuras de datos, sino también por la reacción de los usuarios ante un sistema operativo que cambia constantemente.

La otra dificultad de cualquier algoritmo de evaluación es que modificará el entorno donde se usa el algoritmo.

Los algoritmos de planificación flexibles pueden ser modificados por el administrador del sistema. Durante el tiempo de construcción, arranque o ejecución del sistema operativo, las variables utilizadas por los planificadores se pueden modificar para reflejar el uso esperado del sistema en el futuro.

6.8 Coordinación de procesos

Los sistema concurrentes son un conjunto de procesos: los procesos del **SO** ejecutan código del sistema y los procesos de usuario. Todos estos procesos, pueden, potencialmente, ejecutarse concurrentemente.

6.8.1 Antecedentes

Los procesos productor-consumidos son habituales en los **SO**. Un proceso *productor* genera información que será utilizada por el proceso *consumidor*.

Para que los procesos productores y consumidores puedan ejecutarse concurrentemente, debemos crear un depósito (*pool*) de buffers que pueda llenar el productor y vaciar el consumidor. Un productor puede producir en un buffer mientras que un consumidor consume de otro. Se deben sincronizar el productor y el consumidor, para que este último no trate de consumir algo que aún no se ha producido; en este caso el consumidor deberá esperar a que se produzca algo.

El caso del productor-consumidor con *buffer ilimitado* no establece límites en el número de buffers. Si fuera con *buffer limitado*, el productor deberá esperar si están todos los buffers llenos.

6.8.2 El problema de la sección crítica

Cada segmento tiene un segmento de código, llamado *sección crítica*, en el cual el proceso puede estar modificando variables comunes, actualizando una tabla, etc. La característica importante del sistema es que, cuando un proceso se ejecuta en su sección crítica, no se permite que ningún otro proceso se ejecute en su sección. La ejecución de las secciones críticas de los procesos es *mutuamente excluyente* en el tiempo. El problema de la sección crítica consiste en diseñar un protocolo que los procesos puedan usar para cooperar. Cada proceso debe solicitar permiso para entrar en su sección crítica; la sección de código que implementa esta solicitud es la sección de *entrada*. A la sección crítica puede seguir una sección de *salida* y el código restante es la sección *restante*.

Una solución para el problema de la sección crítica debe cumplir los tres requisitos siguientes:

1. **Exclusión mutua.**
2. **Progreso**: si ningún proceso se está ejecutando en su sección crítica y hay otros procesos que desean entrar en las suyas, entonces sólo aquellos procesos que no se están ejecutando en su sección restante pueden participar en la decisión de cuál será el siguiente en entrar en la sección crítica, y esta selección no puede postergarse indefinidamente.
3. **Espera limitada**: debe haber un límite en el número de veces que se permite que los demás procesos entre en su sección crítica después de que un proceso haya efectuado una solicitud para entrar en la suya y antes de que se conceda esa solicitud.

Debemos suponer que las instrucciones básicas del lenguaje de máquina se ejecutan atómicamente, es decir, si dos de estas instrucciones se ejecutan concurrentemente, el resultado equivale a su ejecución secuencial siguiendo un orden desconocido. Así, si una carga y un almacenamiento se ejecutan concurrentemente, la carga recibirá el valor viejo o el nuevo, pero no una combinación de ambos.

Estructura general de un proceso:

repeat
sección de *entrada*
sección *crítica*
sección *de salida*
ón restante
secci**until** *falso*.

6.8.3 Hardware de sincronización

El problema de la sección crítica puede solucionarse simplemente evitando que ocurran interrupciones mientras se modifica una variable compartida, aunque esto no siempre es factible. Por esto, muchas máquinas ofrecen instrucciones de hardware

especiales que permiten evaluar y modificar el contenido de una palabra o intercambiar atómicamente el contenido de dos palabras.

La característica importante es que la instrucción se ejecuta atómicamente, como una unidad ininterrumpible, de modo que si dos instrucciones *evaluar-asignar* se ejecutan simultáneamente (cada una en una CPU diferente), lo harán secuencialmente siguiendo algún orden arbitrario.

Si la máquina permite la instrucción *evaluar-asignar*, entonces podemos implementar la exclusión mutua declarando una variable booleana *cerradura* con valor inicial *falso*.

6.8.4 Semáforos

Para superar problemas más complejos de la sección crítica, se utiliza una herramienta de sincronización llamada *semáforo*. Un semáforo es una variable entera a la que, salvo por la asignación de valores iniciales, sólo puede accederse mediante dos operaciones atómicas comunes: *espera* (*wait*) y *señal* (*signal*).

Las modificaciones al valor entero del semáforo en las operaciones *espera* y *señal* se ejecutan indivisiblemente.

6.8.4.1 Utilización

Los semáforos pueden utilizarse al tratar el problema de la sección crítica para n procesos; estos n procesos comparten un semáforo común *mutex* (que representa la exclusión mutua), con un valor inicial 1.

Los semáforos también pueden usarse para resolver varios problemas de sincronización. Considere dos procesos que se ejecutan concurrentemente:P1 con un enunciado E1, y P2 con un enunciado E2. Suponga que requerimos que E2 se ejecute sólo después de que E1 haya terminado. Podemos implementar este esquema fácilmente permitiendo que P1 y P2 compartan un semáforo común *sinc*, con un valor inicial 0. Como *sinc* tiene un valor inicial 0, P2 ejecutará E2 únicamente después de que P1 haya invocado a *señal* (*sinc*), que está después de E1.

6.8.4.2 Implementación

La principal desventaja de las soluciones para el problema de sección crítica requiere una espera *activa*. Mientras un proceso se encuentra en su sección crítica, cualquier otro proceso que intente entrar en la suya, deberá ejecutar un ciclo continuo en la sección de entrada. La espera activa desperdicia ciclos de CPU que otros procesos podrían usar productivamente. A este tipo de semáforo se le llama *cerradura giratorio*. La ventaja que tienen es que no es necesario efectuar un cambio de contexto, que puede llevar mucho tiempo, cuando el proceso debe esperar a una cerradura.

El aspecto crucial de los semáforos es que se ejecutan atómicamente. Debemos garantizar que dos procesos no puedan ejecutar *espera* y *señal* al mismo tiempo en el mismo semáforo. Esta situación es un problema de sección crítica y puede solucionarse de dos manera:

En un entorno con una sola CPU, podemos inhibir las interrupciones mientras se ejecutan las operaciones *espera* y *señal*.

En un entorno multiprocesador, es posible intercalar de manera arbitraria instrucciones de procesos diferentes. Si el hardware no ofrece ninguna instrucción especial, podemos emplear cualquier solución correcta implementada en el software para el problema de la sección crítica, donde las secciones críticas consisten en los procedimientos *espera* y *señal*.

6.8.4.3 Bloqueos mutuos y bloqueos indefinidos

Decimos que un conjunto de procesos está en un estado de bloque mutuo cuando cada uno de los procesos del conjunto está esperando un suceso que únicamente puede ser provocado por otro proceso del conjunto. Los sucesos a los que aquí nos referimos son la adquisición y liberación de recursos.

Otro problema relacionado con el bloque mutuo es el *bloque indefinido* o *inanición*, en que los procesos pueden esperar indefinidamente dentro del semáforo. El bloque

indefinido se puede presentar si añadimos y eliminamos procesos de la lista asociada con un semáforo en orden LIFO.

6.8.4.4 Problemas (o errores) de sincronización

Si se utilizan incorrectamente los semáforos, pueden surgir errores de sincronización difíciles de detectar (como dos procesos ejecutándose en su sección crítica). Las dificultades pueden surgir incluso si un solo proceso no se comporta adecuadamente.

6.8.5 Comunicación entre procesos

Muchos de los problemas que se presentan como de sincronización, son en realidad problemas de comunicación. Principalmente hay dos esquemas complementarios de comunicación: sistema de *memoria compartida* y de *mensajes*.

Los sistemas de memoria compartida requieren que los procesos comunicantes compartan algunas variables.

En un sistema de memoria compartida, la responsabilidad de proporcionar la comunicación recae sobre los programadores de aplicaciones, el **SO** sólo tiene que ofrecer la memoria compartida. El método de mensajes permite que los procesos intercambien mensajes y la responsabilidad de proporcionar la comunicación recae sobre el **SO**.

Estos dos esquemas no son mutuamente excluyentes, y podrían usarse simultáneamente.

Un medio para la comunicación entre procesos facilita básicamente dos operaciones: **enviar** (*mensaje*) y **recibir** (*mensaje*).

Los mensajes que envía un proceso pueden ser de tamaño fijo o variable.

Para que dos procesos se comuniquen, debe existir un *enlace de comunicaciones* entre ellos.

Un enlace es unidireccional únicamente cuando cada proceso conectado a él puede enviar o recibir, y cada enlace tiene conectado por lo menos un proceso receptor.

Se cuenta, también con varios métodos para implementar lógicamente un enlace y las operaciones **enviar** y **recibir**: comunicación directa o indirecta, comunicación simétrica o asimétrica, utilización automática o explícita de buffers, envío por copia o por referencia, mensaje de longitud fija o variable.

6.8.5.1 Nominación

Los procesos que desean comunicarse deben tener una manera de referirse unos a otros. Pueden utilizar la *comunicación directa* o la *comunicación indirecta*, que analizaremos a continuación

Comunicación directa

Cada proceso que desea enviar o recibir un mensaje debe nombrar explícitamente al receptor o emisor de la comunicación. En este esquema, las operaciones primitivas **enviar** y **recibir** se definen de la siguiente manera:

enviar (P, mensaje) enviar un mensaje al proceso P.
recibir (Q, mensaje) recibir un mensaje del proceso Q.

Propiedades:

- ☐☐Se establece un enlace automáticamente entre cada par de procesos que quieren comunicarse.
- ☐☐Un enlace está asociado exactamente a dos procesos.
- ☐☐Entre cada par de procesos en comunicación hay exactamente un enlace.
- ☐☐El enlace es bidireccional.
- ☐☐Simetría en el direccionamiento: tanto el emisor como el receptor tienen que nombrar al otro para comunicarse.

Esquema asimétrico (solo el emisor nombra al receptor):

enviar (P, mensaje) enviar un mensaje al proceso P.
recibir (id, mensaje) recibir un mensaje de cualquier proceso. A *id* se le asigna el nombre del proceso con el cual se ha establecido la comunicación.

La desventaja de ambos esquemas es la limitada modularidad de las definiciones de procesos resultantes. Para cambiar el nombre de un proceso puede ser necesario revisar las definiciones de los demás procesos, y hay que encontrar todas las referencias al nombre anterior para cambiarlas por el nuevo. Esta situación no es deseable desde el punto de vista de la compilación por separado.

Comunicación Indirecta

Los mensajes se envían y reciben usando buzones. Un buzón puede considerarse de manera abstracta como un objeto en el que los procesos pueden colocar mensajes y del cual se pueden extraer los mensajes. Cada buzón tiene una identificación única. Las primitivas **enviar** y **recibir** se definen como sigue:

enviar (A, mensaje) enviar un mensaje al buzón A.
recibir (A, mensaje) recibir un mensaje del buzón A.

Propiedades:

- Enlace entre un par de procesos sólo si comparten un buzón.
- Un enlace puede asociarse a más de dos procesos.
- Entre cada par de procesos en comunicación puede haber varios enlaces diferentes.
- El enlace puede ser unidireccional o bidireccional.

Un buzón puede pertenecer a un proceso o al sistema. Si el buzón pertenece a un proceso, entonces hacemos una distinción entre el dueño (que sólo puede recibir mensajes a través de este buzón) y el usuario del buzón (quien sólo puede enviar mensaje al buzón). Como cada buzón tiene un solo dueño, no hay confusión sobre quién debe recibir un mensaje enviado al buzón. Cuando termina un proceso dueño de un buzón, éste desaparece.

Hay varias maneras de designar al dueño y a los usuarios de un buzón en concreto. Una posibilidad es permitir que un proceso declare variables del tipo *buzón*, siendo este proceso el dueño, mientras que cualquier otro que conozca el nombre del buzón, lo podrá utilizar.

Un buzón que pertenece al sistema operativo tiene existencia propia. El **SO** proporciona un mecanismo que permite a un proceso:

- Crear un nuevo buzón.
- Enviar y recibir mensajes mediante el buzón.
- Destruir un buzón.

6.8.5.2 Utilización de buffers

Un enlace tiene cierta capacidad que determina el número de mensajes que puede contener temporalmente. Esta propiedad puede considerarse como una cola de mensajes unidos al enlace. Hay tres maneras básicas de implementar esta cola:

- **Capacidad cero (**sistemas de mensajes sin buffer**)**: el enlace no puede contener ningún mensaje esperando. El emisor debe esperar a que el receptor reciba el mensaje. Los dos procesos se deben sincronizar para que pueda tener lugar la transferencia de un mensaje.
- **Capacidad limitada**: como máximo pueden residir *n* mensajes en ella. Si la cola no está al enviar un nuevo mensaje, se guarda en la cola y el emisor puede continuar su ejecución sin esperar. Si el enlace está lleno, el emisor debe demorarse hasta que haya espacio disponible en la cola.
- **Capacidad ilimitada**: La cola tiene una longitud potencialmente infinita. El emisor nunca se demora.

En los casos de capacidad no nula, un proceso no sabe si un mensaje ha llegado a su destino después de concluir la operación **enviar**. Si esta información es decisiva para los cálculos, el emisor deberá comunicar explícitamente con el receptor para averiguar se éste recibió el mensaje.

Se dice que estos procesos se comunican en forma *asíncrona*.

Casos especiales:

☐☐El proceso que envía el mensaje nunca se demora, Sin embargo, si el receptor no ha recibido el mensaje antes de que el proceso emisor envíe otro, el primer mensaje se pierde. La ventaja de este esquema es que los mensajes de gran tamaño no tienen que copiarse más de una vez. La desventaja principal es que la tarea de programación se complica.

☐☐El proceso que envía un mensaje se demora hasta que recibe una respuesta.

6.8.5.3 Condiciones de excepción

Un sistema de mensajes es particularmente útil en un entorno distribuido, donde los procesos pueden residir en distintas máquinas.

Cuando ocurre un error en un sistema centralizado o distribuido debe llevarse a cabo algún tipo de recuperación del error (manejo de condiciones de excepción). Condiciones de excepción que debe manejar un sistema en el contexto de un esquema de mensajes.

Terminación del proceso

Tanto un receptor como un emisor pueden terminar antes de haber procesado un mensaje. Esta situación dejará mensajes que nunca se recibirán o procesos que esperan un mensaje que nunca se enviará. Dos casos:

1. Un proceso receptor P puede esperar un mensaje de un proceso Q que ha terminado. Si no se emprende ninguna acción, P quedará eternamente. En este caso el sistema puede terminar la ejecución de P o notificar a P que Q ha terminado.
2. El proceso P puede enviar un mensaje a un proceso Q que ha terminado. En el esquema de los buffers automáticos no hay daños. En el caso sin buffers, P quedará bloqueado para siempre. Como en el caso 1, el sistema puede terminar la ejecución de P o notificar a P que Q ha terminado.

Mensajes perdidos

Un mensaje de un proceso a otro puede perderse en algún lugar de la red de comunicaciones. Existen tres métodos para tratar esto:

1. El **SO** es responsable de detectar el suceso y reenviar el mensaje.
2. El proceso emisor es responsable de detectar este suceso y retransmitir el mensaje, si así lo desea.
3. El **SO** es responsable de detectar el suceso y notificar al proceso emisor que ha perdido el mensaje.

No siempre es necesario detectar los mensajes perdidos.

El método más habitual es utilizar *tiempos límite*. Cuando se envía un mensaje, siempre se devuelve un mensaje de respuesta que confirma su recepción. Entonces, el sistema operativo o un proceso pueden especificar un intervalo de tiempo durante el cual espera la llegada del mensaje de confirmación. Si este intervalo transcurre sin que llegue la respuesta, el **SO** (o el proceso) puede suponer que el mensaje se ha perdido y se reenvía el mensaje.

Mensajes alterados

El mensaje puede llegar a su destino, pero alterarse por el camino. El **SO** retransmitirá el mensaje original o notificará este suceso al proceso. Para detectar este tipo de error generalmente se utiliza la suma de verificación (paridad o CRC).

7. Administración de Memoria

La selección de un esquema de administración de memoria depende de muchos factores, especialmente del diseño del *hardware* del sistema. Cada algoritmo requiere su propio apoyo de hardware.

Los algoritmos de administración de memoria tienen un requisito básico: todo el programa del proceso debe encontrarse en la memoria física. Esta restricción limita el tamaño máximo del proceso al tamaño de la memoria física.

7.1 Antecedentes

La memoria es un gran arreglo de palabras o bytes, cada uno con su propia dirección.

La unidad de memoria solo ve un flujo de direcciones de memoria.

7.1.1 Enlace de direcciones

Para ejecutar un proceso, éste debe cargarse en memoria. El conjunto de procesos en disco que espera entrar en la memoria para ejecutarse integran la *cola de entrada*.

El procedimiento normal consiste en seleccionar uno de los procesos de la cola de entrada y cargarlo en memoria. Esta técnica en ocasiones provoca la relocalización de direcciones o el enlace de referencias externas, según sea necesario.

Un proceso de usuario puede residir en cualquier parte de la memoria física. En la mayoría de los casos, un programa de usuario pasará por varias etapas antes de ejecutarse. En estas etapas las direcciones pueden representarse de distintas maneras. En un programa fuente las direcciones son generalmente simbólicas. A su vez, el editor de enlace o el cargador enlazará estas direcciones relocalizables con direcciones absolutas. Cada enlace es una correspondencia entre un espacio de direcciones y otro.

El *enlace* de las instrucciones y datos con las direcciones de memoria casi siempre puede efectuarse en cualquier etapa del camino:

- **Compilación**: si en el momento de la compilación se sabe donde residirá el programa en memoria puede generarse *código absoluto*. (los archivos como son de código absoluto).
- **Carga**: si en el momento de la compilación no se conoce donde residirá el programa, entonces el compilador deberá generar código *relocalizable*. En este caso se posterga el enlace final hasta el momento de la carga.
- **Ejecución**: si durante la ejecución de un proceso puede moverse de un segmento a otro, entonces el enlace final se debe postergar hasta el momento de la ejecución. Se debe tener hardware especial para que este esquema funcione.

7.1.2 Carga dinámica

Para obtener una mejor utilización del espacio de memoria podemos utilizar *la carga dinámica*, con la cual una rutina no se carga hasta que se la llama. Todas las rutinas se almacenan en disco en formato de carga relocalizable. El programa principal se carga en memoria y se ejecuta. Cuando una rutina tiene que llamar a otra, la rutina que llama primero comprueba si se ha cargado la otra. Si no es así, se llama al cargador de enlace relocalizable para que se cargue en memoria la rutina deseada y actualice las tablas para reflejar este cambio. Entonces el control se transfiere a la rutina recién cargada.

La ventaja de la carga dinámica es que nunca se carga una rutina que no se usa. Este esquema es particularmente útil cuando se requieren grandes cantidades de código para manejar situaciones que ocurren con poca frecuencia (rutinas de error, por ej.).

La carga dinámica no requiere apoyo especial del **SO**. Es responsabilidad de los usuarios diseñar sus programas para aprovechar este esquema.

7.1.3 Enlace dinámico

La mayoría de los **SO** sólo permiten el enlace estático, donde las bibliotecas de lenguaje del sistema son tratados como cualquier módulo objeto y el cargador las combina con el contenido del programa binario. En el enlace dinámico, en vez de postergar la carga hasta la ejecución, se posterga el enlace. Esta característica se emplea generalmente con las bibliotecas del sistema, como las bibliotecas de subrutinas de lenguajes. Sin este recurso, todos los programas de un sistema necesitan incluir en su contenido ejecutable una copia de la biblioteca de su lenguaje. Con el enlace dinámico, en el contenido binario se incluye un *fragmento* (*stub*) para cada referencia a una rutina de la biblioteca. Este fragmento es un pequeño trozo de código que indica cómo localizar la rutina de biblioteca residente en memoria. Al ejecutar este fragmento se reemplaza a sí mismo con la dirección de la rutina y la ejecuta, quedando ya establecida la dirección. En este esquema, todos los procesos que utilizan la biblioteca de lenguaje ejecutan sólo una copia del código de la biblioteca.

Esta característica puede extenderse para incluir actualizaciones a bibliotecas. Una nueva versión puede sustituir a una biblioteca y todos los programas hacen referencia a la biblioteca usarán automáticamente la nueva versión. Para que los programas no ejecuten accidentalmente versiones nuevas e incompatibles de las bibliotecas, tanto en el programa como en la biblioteca se incluye información de la versión. Se pueden cargar distintas versiones de una misma biblioteca en memoria.

7.1.4 Superposiciones

Para que un proceso pueda ser mayor que la cantidad de memoria que se le asigna, en ocasiones se utiliza una técnica llamada *superposiciones*; la idea es conservar en memoria sólo aquellas instrucciones y datos que se requieren en un momento determinado. Cuando se necesitan otras instrucciones, se cargan en el espacio que antes ocupaban las que ya no se requieren.

Para construir las superposiciones se necesitan algoritmos especiales de relocalización y enlace.

No requieren ningún apoyo especial del **SO**. El programador es el que tiene que diseñar y programar adecuadamente la estructura de las superposiciones. Ésta puede ser una tarea de gran magnitud que requiere un conocimiento total de la estructura del programa, su código y sus estructuras.

7.2 Intercambios

Un proceso puede *intercambiarse* temporalmente saliendo de la memoria a un *almacenamiento secundario*, y regresando luego a la memoria para continuar su ejecución. Idealmente, el administrador de la memoria puede intercambiar procesos con la velocidad suficiente para que siempre haya procesos en memoria, listos para ejecutarse, cada vez que el planificador de la CPU quiere volver a planificarla. El quantum debe tener el tamaño suficiente para que se efectúe una cantidad de cálculos razonable entre intercambios.

Para algoritmos basados en prioridades: si llega un proceso de mayor prioridad y desea ejecutarse, el administrador de memoria puede intercambiarlo con un proceso de menor prioridad para así cargar y ejecutar el de mayor prioridad. Cuando éste termina, puede devolver el proceso de menor prioridad y continuar su ejecución. Esta variante se denomina *salida y entrada por intercambio*.

Normalmente un proceso que sale de un intercambio regresará al mismo espacio de memoria que antes ocupaba. Esta restricción la determina el método de enlace de direcciones.

Los intercambios requieren *almacenamiento auxiliar*. Debe tener el tamaño suficiente para contener copias de la imagen de memoria de todos los usuarios y ofrecer acceso directo a esta imagen. El sistema mantiene una *cola de procesos listos* que consiste en todos los procesos cuya imagen de memoria se encuentra en el almacenamiento auxiliar o en memoria y están listos para ejecutarse. Cuando el planificador de la CPU decide ejecutar un proceso, llama al despachador, el cual comprueba si el siguiente proceso de la cola está en memoria. Si no está, y no se cuenta

con una región de memoria libre, el despachador intercambia un proceso en memoria con el proceso deseado. Obviamente el tiempo de cambio de contexto en un sistema de intercambios de este tipo es bastante alto.

La mayor parte del tiempo de intercambio se invierte en la transferencia, que es directamente proporcional a la *cantidad* de memoria que se intercambia.

Se puede intercambiar sólo la cantidad de memoria que se utiliza, reduciendo el tiempo de intercambio. Para que este esquema sea efectivo, el usuario debe informar al **SO** de cualquier cambio en los requisitos de memoria.

Existen otras restricciones: si queremos intercambiar un proceso, debemos estar seguros de que está completamente inactivo. Cualquier solicitud de E/S que está pendiente tiene una importancia especial. Si la E/S tiene acceso asíncrono a la memoria del usuario para usar buffers, e intercambiamos el proceso, la E/S podría modificar al nuevo proceso intercambiado.

Las dos soluciones a este problemas son: nunca intercambiar un proceso con E/S pendiente o, usar sólo buffers del **SO** para E/S.

7.3 Asignación de una sola partición

Esquema más sencillo, al usuario se le proporciona la máquina básica y tiene control absoluto sobre todo el espacio de memoria.

Ventajas: Máxima flexibilidad al usuario, sencillo, costo mínimo, no requiere hardware especial, ni software del **SO**.

Desventajas: el **SO** no ofrece servicios, no hay control sobre las interrupciones, ni mecanismos de procesamiento de llamadas al sistema, o errores, ni espacio para la multiprogramación.

Este enfoque es sólo para sistemas dedicados donde se requiere flexibilidad.

El siguiente esquema más sencillo consiste en dividir la memoria en dos: 1-Usuario 2- **SO** residente. Es posible alojarlo en la memoria alta, o baja. En este esquema, debemos proteger el código y los datos del **SO** frente a los cambios provocados por el proceso de usuario. Esta protección debe ser proporcionada por el hardware.

Otro problema que hay que tener en cuenta es la carga de los procesos de usuario. La primer dirección del programa de usuario, será la que sigue al valor del registro base. Si durante la compilación se conoce la dirección base, puede generarse código absoluto, pero si luego cambia, habría que recompilar. Como alternativa, el compilador puede generar código relocalizable.

Un problema con este esquema es que el valor base debe permanecer *estático* durante la ejecución de un programa. Si las direcciones de usuario están enlazadas a las direcciones físicas por medio de la base, entonces estas direcciones no serán válidas si la base cambia. Por esto, la dirección base sólo puede cambiar cuando no se ejecuta ningún programa de usuario. Existen casos donde es deseable cambiar el tamaño del **SO**.

Hay dos métodos para modificar el esquema básico para permitir que cambie dinámicamente el tamaño del sistema operativo:

1. Cargar el proceso de usuario en la memoria alta, descendiendo hacia el valor del registro base. La ventaja es que todo el espacio no utilizado se encuentra en la parte media, por lo que, de ser necesario, el proceso de usuario o el **SO** pueden extenderse a la memoria no utilizada.
2. Postergar el enlace de direcciones hasta la *ejecución*. Requiere un apoyo de hardware ligeramente distinto. Al registro base ahora se lo llama registro de *relocalización*. El valor del registro base se *suma* a cada dirección generada por un proceso de usuario en el momento de enviarla a memoria. Observe que un programa de usuario nunca ve las direcciones físicas reales, trata con direcciones *lógicas*. El hardware de correspondencia de memoria convierte las direcciones lógicas en direcciones físicas. Un cambio en la localidad de inicio sólo requiere un cambio en el registro base y la transferencia de toda la memoria del usuario a las localidades correctas respecto al nuevo valor base. Toda la información que el proceso de usuario pasa al **SO** debe ser relocalizada explícitamente por el software del **SO** antes de usarse.

7.4 Asignación de particiones múltiples

En un sistema multiprogramado, el problema de la administración de memoria consiste en asignar memoria a los distintos procesos que esperan en la cola de entrada para ser transferidos a memoria.

Uno de los esquemas más sencillos consiste en dividirla en varias *particiones* de tamaño fijo. Cada partición puede contener exactamente un proceso. El nivel de multiprogramación está limitada está limitado por el número de particiones. Cuando una partición está libre, se selecciona un proceso en la cola de entrada y se carga en la partición libre.

7.4.1 Esquema básico

El **SO** conserva una tabla que indica cuáles partes de la memoria están disponibles u cuáles están ocupadas. Inicialmente toda la memoria está disponible para los procesos de usuario. Cuando llega un proceso y necesita memoria, se busca un hueco de tamaño suficiente para ese proceso. Si lo encontramos, sólo asignamos la cantidad de memoria necesaria, quedando disponible el resto para satisfacer solicitudes ulteriores.

En cualquier momento hay *un conjunto* de huecos de distintos tamaños y dispersos por toda la memoria. Cuando llega u proceso y necesita memoria, buscamos en este conjunto un hueco con el tamaño suficiente para el proceso. Si el hueco es demasiado grande, se divide en dos: una parte se asigna al proceso que llega y la otra se devuelve al conjunto de huecos. Cuando termina un proceso, libera su bloque de memoria, el cual se coloca de nuevo en el conjunto de huecos. Si el nuevo hueco es adyacente a otros, los fusionamos formando uno mayor. Al llegar a este punto, necesitaríamos comprobar si hay procesos esperando memoria y si esta nueva memoria liberada y recombinada puede satisfacer las solicitudes de algunos de estos procesos.

Se trata ver cómo satisfacer una solicitud de tamaño *n* a partir de una lista de huecos libres.

Existen muchas soluciones para este problema:

Primer ajuste: asignar el *primer* hueco que tenga el tamaño suficiente

Mejor ajuste: asignar el hueco *más pequeño* que tenga el tamaño suficiente.

Peor ajuste: asignar el hueco *más grande*, produciendo el hueco sobrante más grande.

Tanto el primer como el mejor ajuste son mejores que el pero ajuste en cuanto a la reducción del tiempo y de la utilización del almacenamiento (el primer ajuste es más rápido).

Estos algoritmos *padecen fragmentación externa*.

Fragmentación Externa: se presenta cuando el espacio de memoria es suficiente para atender una solicitud pero no es contiguo.

Regla del 50 por ciento: dados *N* bloques asignados, se perderán otros *0.5N* por fragmentación.

Como varios procesos pueden residir en memoria al mismo tiempo, ésta se debe proteger. Podemos proporcionar esta protección utilizando registros base y límite. El registro base contiene el valor de la menor dirección física; el registro límite contiene el intervalo de direcciones lógicas; relocalizamos *dinámicamente* la dirección lógica sumando el valor del registro base y esta dirección relocalizada se envía a memoria.

Cuando el planificador de la CPU selecciona un proceso, el despachador carga los valores correctos en los registros base y límite. Como cada dirección generada por la CPU se compara con estos registros, podemos proteger a los demás programas y datos de usuario contra modificaciones por la ejecución de este proceso.

Fragmentación Interna: diferencia entre la memoria solicitada y la asignada.

A veces, al asignar a un proceso un hueco de memoria, puede haber una muy pequeña porción de memoria sobrante, que requerirá procesamiento adicional mayor que el propio hueco, por lo tanto se asigna este pequeño hueco al proceso, por más que no lo utilice.

7.4.2 Planificación a largo plazo

El planificador a largo plazo tiene en cuenta los requisitos de memoria de cada proceso y la cantidad de memoria disponible para determinar a qué procesos se les asigna.

En cualquier momento tenemos una lista de tamaños de bloques disponibles y la cola de entrada. La memoria se asigna a los procesos hasta que finalmente no pueden satisfacerse los requisitos de memoria del siguiente proceso. Entonces el planificador a largo plazo puede esperar hasta que esté disponible un bloque de tamaño suficiente o recorrer la cola de entrada para ver si pueden satisfacerse las solicitudes de menor cantidad de memoria de algún proceso de menor prioridad.

Con este esquema casi no hay fragmentación interna, ya que las particiones se crean con el tamaño solicitado por el proceso; sin embargo, puede haber fragmentación externa.

En el peor de los casos, podemos tener un bloque de memoria libre (desperdiciado) entre cada dos procesos. Si toda esta memoria estuviera en un gran bloque libre, podríamos ejecutar varios procesos más. La selección entre el primer ajuste y el mejor puede afectar la cantidad de fragmentación. Otro factor es de cuál extremo del bloque libre se toma la memoria a asignar.

7.4.3 Compactación

Una solución al problema de la fragmentación externa es la *compactación*. El objetivo consiste en desplazar el contenido de la memoria para colocar junta toda la memoria libre en un solo bloque de gran tamaño.

La compactación no siempre es posible. Si la relocalización es estática y se realiza durante la compilación (o ensamblado) o la carga, la compactación no puede realizarse.

Si las direcciones se relocalizan dinámicamente, esta relocalización sólo requiere mover el programa y los datos, y luego cambiar el registro base para reflejar la nueva dirección base.

Cuando la compactación es posible, debemos determinar su costo. El algoritmo de compactación más sencillo consiste en mover todos los procesos hacia un extremo de la memoria; todos los huecos se mueven en la dirección contraria produciendo un gran hueco de memoria disponible. Este esquema puede ser bastante costoso.

También se puede combinar el intercambio con la compactación. Un proceso puede descargarse de memoria principal por intercambio al almacenamiento auxiliar y reincorporarse más tarde. Cuando el proceso tiene que reincorporarse por intercambio, pueden aparecer varios problemas. Si se utiliza la relocalización estática, el proceso debe reincorporarse a las mismas direcciones de memoria que antes ocupaba.

Si se emplea la relocalización dinámica, entonces el proceso se puede reincorporar a una localidad diferente. En este caso encontramos un bloque libre, compactando si es necesario, y reincorporamos el proceso.

Una estrategia para la compactación consiste en descargar los procesos que hay que mover, y reincorporarlos a localidades de memoria diferentes.

7.5 Registros base múltiples

Una forma de reducir la cantidad de fragmentación externa es dividir en varias partes la memoria que necesita un proceso. Cada parte es menor al todo y, por tanto, más fácil de acomodar en la memoria. Para esto hay que proporcionar múltiples registros base con un mecanismo para traducir las direcciones lógicas a físicas.

Una forma de lograr esto es dividir la memoria en dos partes disjuntas. El sistema tiene dos pares de registros base y límite. La memoria se divide por la mitad usando el bit de dirección de orden superior. La memoria baja se relocaliza y limita usando el par 0 de registros base y límite; para la parte alta se usa el par 1. Por convención, los compiladores y ensambladores colocan valores de sólo lectura (como constantes e instrucciones) en la memoria alta y las variables en la memoria baja. Se asocian bits de protección a cada par de registros y pueden asegurar que la memoria alta sea de sólo lectura. Esta disposición permite compartir programas (almacenados como de sólo lectura en la memoria alta) entre varios procesos de usuario, cada uno con su propio segmento en memoria baja.

Otra manera de lograr esto, es separar un programa en dos partes: código y datos. La CPU sabe si quiere una instrucción o datos. Por tanto se proporcionan dos pares de registros base y límite (uno para instrucciones, otro para datos). El par para instrucciones

es, automáticamente, de sólo lectura, de manera que los programas se puedan compartir entre distintos usuarios.

Al separar programas y datos, y relocalizarlos por separado, podemos compartir programas entre varios usuarios; así utilizamos mejor la memoria, reduciendo tanto la fragmentación como las copias múltiples del mismo código.

7.6 Paginación

La *paginación* permite que la memoria de un proceso no sea contigua, y que a un proceso se le asigne memoria física donde quiera que ésta esté disponible. La paginación evita el gran problema de acomodar trozos de memoria de tamaño variable en el almacenamiento auxiliar.

7.6.1 Hardware

La memoria física se divide en bloques de tamaño fijo llamados *marcos*. La memoria lógica también se divide en bloques del mismo tamaño llamados *páginas*. Cuando un proceso se va a ejecutar, sus páginas se cargan desde el almacenamiento auxiliar en cualesquiera de los marcos disponibles. El almacenamiento auxiliar se divide en bloques de tamaño fijo del mismo tamaño que los marcos de memoria.

Cada dirección generada por la CPU se divide en dos partes: un *número de página* (p) y un *desplazamiento en la página* (d). El número de página se utiliza como índice en una *tabla de páginas*; que contiene la dirección base para cada página de la memoria física. La dirección base se combina con el desplazamiento en la página para definir la dirección de memoria física que se envía a la unidad de memoria. El tamaño de la página, al igual que el tamaño del marco, está definido por el hardware.

La paginación es una forma de relocalización dinámica. Cada dirección lógica está enlazada a la dirección física mediante el hardware de paginación.

7.6.2 Planificación a largo plazo

El planificador comprueba la memoria disponible, que se representa como una lista de marcos no asignados. Cada página de usuario necesita un marco; por tanto, si el proceso requiere *n* páginas, debe haber *n* marcos disponibles en memoria; si los hay, el planificador a largo plazo los asigna al proceso. La primera página del proceso se carga en uno de los marcos asignados y el número de marco se registra en la tabla de páginas para este proceso; la siguiente página se carga en otro marco, y su número de marco se coloca en la tabla de páginas.

Cuando usamos un esquema de paginación no tenemos fragmentación externa: *cualquier* marco libre se puede asignar a un proceso que lo necesite. Sin embargo, podemos tener cierta fragmentación interna.

Son deseables los tamaños de páginas pequeños, pero cada entrada de tabla de páginas implica bastante gasto adicional, que disminuye al aumentar el tamaño de la página.

Cada sistema operativo tiene sus propios métodos para almacenar tablas de páginas. La mayoría asigna una tabla de páginas para cada proceso. En el bloque de control del proceso se almacena un apuntador a la tabla de páginas junto con los valores de registros. Cuando al despachador se le indica que inicie un proceso, debe volver a cargar los registros del usuario y definir los valores correctos de hardware para la tabla de páginas usando la tabla de páginas de usuario almacenada.

7.6.3 Implantación de la tabla de páginas

Puede lograrse de varias maneras. El caso más sencillo, la tabla de páginas se implanta como un conjunto de registros dedicados. Estos registros deben construirse usando lógica de muy alta velocidad para que la traducción de direcciones de páginas sea muy eficiente. El despachador de la CPU vuelve a cargar estos registros de la misma manera que carga los demás. Por supuesto, las instrucciones para cargar o modificar los registros de la tabla de páginas son privilegiadas.

La utilización de registros para la tabla de páginas es satisfactoria si ésta es razonablemente pequeña (por ejemplo, 246 entradas). Sin embargo, la mayoría de los

computadores actuales permiten que la tabla de páginas sea muy grande. En estos casos, la tabla de páginas se conserva en memoria principal y un *registro base de tabla de páginas* (PTBR) apunta a la tabla de páginas. Para cambiar entre tablas de páginas sólo hay que modificar este registro, lo que reduce en gran medida el tiempo de cambio de contexto.

El gran problema es el tiempo requerido para acceder a una localidad de memoria: Si queremos llegar a la localidad *i*, primero debemos utilizar el índice de la tabla de páginas, usando el valor de desplazamiento del PTBR para el número de página *i*. Esta tarea requiere acceso a memoria. Nos proporciona el número de marco, el cual se combina con el desplazamiento en la página para producir la dirección real. Luego podemos acceder al lugar deseado de la memoria. Este procedimiento realiza dos accesos a memoria, lo cual lo hace lento.

La solución común para este problema, consiste en utilizar una memoria especial, de tamaño pequeño, llamada *registros asociativos* o *buffers de traducción* con búsqueda anticipada (TLB). Un conjunto de registros asociativos se construye con memoria de muy alta velocidad; cada registro consta de dos partes: una clave y un valor, y cuando se presenta un elemento a los registros asociativos, se compara simultáneamente con todas las claves. Si el elemento se encuentra, se devuelve el campo de valor correspondiente. La búsqueda es muy rápida, pero el hardware es muy costoso.

Los registros asociativos se utilizan con las tablas de páginas de la siguiente manera: los registros asociativos contienen sólo algunas de las entradas de la tabla de páginas. Cuando la CPU genera una dirección lógica, su número de página se presenta a un conjunto de registros asociativos que contienen números de página y sus números de marco correspondiente. Si el número de página se encuentra en los registros asociativos, su número de marco está inmediatamente disponible y se utiliza para acceder a la memoria.

Si el número de página no se encuentra en los registros asociativos, hay que efectuar una referencia de memoria a la tabla de páginas. Cuando se obtiene el número de marco, podemos usarlo para tener el acceso deseado a memoria. Además, añadimos los números de página y marco a los registros asociativos, para que puedan localizarse con rapidez en la siguiente referencia.

Tasa de aciertos: porcentaje de ocasiones que se encuentra un número de página en los registros asociativos.

La tasa de aciertos está evidentemente relacionada con el número de registros asociativos.

7.6.4 Páginas compartidas

Otra ventaja de la paginación es la posibilidad de *compartir* código común, de especial importancia en un entorno de tiempo compartido.

Código reentrante (o Código Puro): código que no se modifica a sí mismo.

Si el código es reentrante, dos o más procesos pueden ejecutar el mismo código al mismo tiempo. Cada proceso tiene su propia copia de los registros y su almacenamiento para contener los datos durante la ejecución. Los datos variarán para cada proceso.

7.6.5 Protección

La protección de memoria en un entorno paginado se logra por medio de bits de protección asociados a cada marco. Normalmente, estos bits se conservan en la tabla de páginas. Un bit puede definir si una página es de sólo lectura y escritura. Cada referencia a la memoria pasa por la tabla de páginas para encontrar el número de marco correcto. Al mismo tiempo que se calcula la dirección física, puede consultarse los bits de protección para verificar que no se efectúen escrituras en una página de sólo lectura. Podemos crear hardware para brindar protección de sólo lectura, lectura y escritura o de sólo ejecución. O, proporcionando bits de protección separados para cada tipo de acceso, permitiendo cualquier tipo de combinación.

Es poco común que un proceso utilice todo su intervalo de direcciones. En estos casos sería un derroche crear una tabla de páginas con registros para cada página del intervalo de direcciones. Algunos sistemas ofrecen hardware que indica el tamaño de la

tabla de páginas, a través de un registro de longitud de tabla de páginas (PTLR). Este valor se compara con cada dirección lógica para asegurar que la dirección se encuentre en el intervalo válido para el proceso.

7.6.6 Dos perspectivas de la memoria

Un aspecto importante de la paginación es la clara separación entre la perspectiva del usuario e la memoria y la memoria física real. En realidad, el programa de usuario está disperso por la memoria física. La diferencia entre la perspectiva que el usuario tiene de la memoria y memoria física real se concilia a través del hardware de traducción de direcciones. Las direcciones lógicas se traducen a direcciones físicas. Esta correspondencia se oculta al usuario y es controlada por el **SO**.

Las direcciones físicas y lógicas pueden ser diferentes. Con la multiprogramación, el sistema puede utilizar toda la memoria; sin embargo, los usuarios no pueden usar más memoria que antes, ya que el espacio de direcciones lógicas no ha aumentado.

El sistema operativo controla la correspondencia de memoria física y lógica. Puesto que éste administra la memoria física, debe estar al tanto de: qué marcos están asignados, qué marcos están libres, cuántos hay en total, etc. Esta información casi siempre se conserva en una estructura de datos llamada *tabla de marcos*, la cual tiene una entrada por cada marco físico de página, que indica si el marco está libre o asignado, y de estar asignado, a qué pagina de qué proceso.

El **SO** mantiene una copia de la tabla de páginas para cada usuario, al igual que mantiene una copia de registros. Esta copia se emplea para traducir las direcciones lógicas a direcciones físicas reales cada vez que el **SO** debe transformar manualmente una dirección lógica en física. También la utiliza el despachador de la CPU para definir la tabla de páginas del hardware cuando se asigna un proceso a la CPU.

7.7 Segmentación

7.7.1 Perspectiva de memoria del usuario

El programador o usuario de un sistema, no considera la memoria como un arreglo lineal de palabras; más bien prefiere pensar en ella como un conjunto de segmentos de tamaño variable, sin ningún orden en especial.

La *segmentación* es un esquema de administración de memoria que apoya la perspectiva que el usuario tiene de la memoria. Un espacio de direcciones lógicas se compone de un conjunto de segmentos, cada uno de los cuales tiene un nombre y una longitud. Las direcciones especifican el nombre del segmento y el desplazamiento dentro de él, de manera que el usuario especifica cada dirección con dos cantidades: el nombre del segmento y un desplazamiento.

Para hacer más sencilla la implantación, los segmentos se numeran y se hace referencia a ellos mediante un número de segmento, en vez de por un nombre.

7.7.2 Hardware

Debemos definir una forma de implantar la correspondencia entre direcciones bidimensionales definidas por el usuario y direcciones físicas unidimensionales. Esta correspondencia se logra por medio de una *tabla de segmentos*.

Una dirección lógica consiste en dos partes: un número de segmentos (*s*) y un desplazamiento dentro del segmentos (*k*). El número de segmento se utiliza como índice en la tabla de segmentos. Cada entrada de la tabla de segmentos tiene una *base* y un *límite* para el segmento. El desplazamiento *d* de la dirección lógica debe estar entre 0 y el límite del segmento.

Si el desplazamiento es válido, se suma a la base del segmento para producir la dirección en memoria física de la palabra deseada. La tabla de segmentos es, en esencia, un arreglo de pares de registros base y límite.

7.7.3 Implantación de las tablas de segmentos

La segmentación es concepto complejo. Al igual que en una tabla de páginas, una tabla de segmentos puede ubicarse en registros rápidos o en memoria. Se pueden

efectuar referencias rápidas a una tabla de segmentos almacenadas en registros; para ahorrar tiempo, la suma a la base y la comparación con el límite pueden efectuarse simultáneamente.

Un registro *base de la tabla de segmentos* (STBR) apunta a la tabla de segmentos. Así mismo, ya que el número de segmentos que utiliza un programa puede variar considerablemente, se usa un registro de *longitud de la tabla de segmentos* (STLR). Para una dirección lógica (*s*, *d*), primeros comprobamos si el número de segmentos *s* es legal (es decir, *s*<STLR). Luego sumamos el número de segmentos al STBR, lo que da como resultado la dirección en memoria de la entrada de la tabla de segmentos. Esta entrada se lee de la memoria y continuamos como antes: comparamos el desplazamiento con la longitud del segmento y calculamos la dirección física de la palabra deseada como la suma de la base del segmento más el desplazamiento.

Esta transformación requiere dos referencias a memoria por cada dirección lógica. La solución normal consiste en emplear un conjunto de registros asociativos para que contengan las entradas usadas más recientemente.

7.7.4 Protección y compartimiento

Una ventaja particular de la segmentación es la asociación de la protección a los segmentos. Como los segmentos representan una porción de un programa definida semánticamente, es probable que todos los registros del segmento se utilicen de la misma manera. En una arquitectura moderna, las instrucciones no se modifican a sí mismas, por lo que los segmentos de instrucciones pueden definirse de sólo lectura o de sólo ejecución. Si colocamos un arreglo en su propio segmento, el hardware de administración de memoria verificará automáticamente que los índices del arreglo sean legales y no salgan de los límites del arreglo. De esta manera, el hardware detectará muchos errores comunes en los programas antes que puedan causar daños serios.

Otra ventaja de la segmentación es el *compartimiento* de código o datos. Cada proceso tiene una tabla de segmentos asociada a su bloque de control de proceso, utilizada por el despachador para definir la tabla de segmentos del hardware cuando se asigna la CPU a este proceso. Los segmentos se comparten cuando las entradas de las tablas de segmentos de dos procesos distintos apuntan a las mismas localidades físicas.

El compartimiento tiene lugar a nivel de segmento, de manera que cualquier información puede compartirse si está definida en un segmento.

Existen algunas consideraciones sutiles. Los segmentos de código con frecuencia contienen referencias a sí mismos.

Los segmentos de sólo lectura (sin apuntadores) pueden compartirse con números de segmento distintos, al igual que muchos segmentos de código que no se referencien a sí mismos de manera directa, sólo indirecta.

7.7.5 Fragmentación

Esta situación es similar a la que se presenta en la paginación, *excepto* que los segmentos tienen longitud *variable*.

La segmentación puede entonces ocasionar fragmentación externa, cuando los bloques de memoria libre son muy chicos para contener un segmento. Una posibilidad es compactar, para juntar los huecos de memoria.

En general, si el tamaño promedio de los segmentos es pequeño, la fragmentación externa suele ser pequeña.

No existe fragmentación interna.

7.8 Segmentación paginada

Es posible combinar estos dos esquemas para mejorar cada uno de ellos.

En el sistema MULTICS las direcciones lógicas se forman a partir de un número de segmento de 18 bits y u desplazamiento de 16 bits (34 bits en total). El número variable de segmentos representa naturalmente a un STLR. Cómo sólo necesitamos tantas entradas en la tabla de segmentos como segmentos hay, en la tablas de segmentos no habrá entradas vacías.

Sin embargo, con segmentos de palabras de 64K, el tamaño promedio de los segmentos podría ser grande y la fragmentación externa constituiría un problema, además del tiempo de búsqueda para asignar un segmento, utilizando el primer o el mejor ajuste, sería excesivo.

La solución que se adoptó fue paginar los segmentos. La paginación elimina la fragmentación externa y hace de la asignación un problema trivial: se puede usar cualquier marco vacío para una página deseada. Observe que la diferencia entre esta solución y la segmentación pura consiste en que la entrada de la tabla de segmentos no contiene la dirección base del segmento, sino la dirección base de una *tabla de páginas* para ese segmento. El desplazamiento en el segmento se divide entonces en un número de 6 bits y un desplazamiento en la página de 10 bits (16 bits en total). El número de página se usa como índice para la tabla de páginas a fin de obtener el número de marco. Por último, el número de marco se combina con el desplazamiento en la página para formar una dirección física.

Ahora debemos tener una tabla de página particular para cada segmento. Sin embargo, como cada segmento tiene limitada su longitud por su entrada de la tabla de segmentos, la tabla de páginas no necesita ser de tamaño completo. Sólo requiere tantas entradas como las que realmente se necesitan. Además, la última página de cada segmento en general no estará llena, por lo que tendremos en promedio media página de fragmentación *interna* por segmento. En consecuencia, aunque hemos eliminado la fragmentación externa, hemos introducido la fragmentación interna y aumentado el espacio consumido por la tabla.

MULTICS pagina la tabla de segmento. Así, generalmente una dirección en MULTICS utiliza un número de segmento para definir un índice de página en una tabla de páginas para la tabla de segmentos. A partir de esta entrada, localiza la parte de la tabla de segmentos que contiene la entrada para este segmento. La entrada de la tabla de segmentos apunta a una tabla de páginas para este segmento, la cual especifica el marco que contiene la palabra deseada.

8. Memoria Virtual

La *memoria virtual* es una técnica que permite la ejecución de procesos que pueden no estar completamente en memoria. La principal ventaja evidente de este esquema es que los programas pueden ser mayores que la memoria física. Además, abstrae la memoria principal para formar un arreglo uniforme muy grande de almacenamiento. Esto libera a los programadores de la preocupación por las limitaciones del almacenamiento en memoria.

8.1 Motivación

Los algoritmos de administración de memoria son necesarios por un requisito básico: todo el espacio de direcciones lógicas de un proceso debe encontrarse en memoria física antes que el proceso se pueda ejecutar. Esta restricción parece necesaria y razonable, pero es lamentable, ya que limita el tamaño de un programa al tamaño de la memoria física.

De hecho, al examinar programas reales nos percatamos que, en muchos casos, no se requiere el programa completo, por ejemplo:

- A los arreglos, listas y tablas frecuentemente se les asigna más memoria de la que realmente necesitan.
- Ciertas opciones y características de un programa rara vez se usan.

La capacidad de ejecutar un programa que se encuentra parcialmente en memoria tendría varias ventajas:

- Un programa ya no está restringido por la cantidad de memoria física disponible.
- Como cada programa de usuario ocuparía menos memoria física, podrían ejecutarse más programas al mismo tiempo, aumentando la utilización de CPU y la productividad, pero sin incrementar el tiempo de respuesta o el de retorno.
- Se requeriría menos E/S para cargar o intercambiar cada uno de los programas de usuario, por lo que se ejecutarían más rápido.

La *memoria virtual* es la separación de la memoria lógica del usuario de la memoria física. Facilita las tareas de programación. Una consecuencia es que las superposiciones casi desaparecen.

Generalmente se implanta mediante *paginación por demanda*.

8.2 Paginación por demanda

Un sistema de paginación por demanda es similar a un sistema de paginación con intercambios. Los procesos residen en memoria secundaria. Cuando queremos ejecutar un proceso, lo metemos en la memoria. Sin embargo, en vez de intercambiar todo el proceso hacia la memoria, utilizamos un intercambiador (paginador en realidad) "perezoso". Un intercambiador perezoso reincorpora una página a memoria a menos que se necesite.

Cuando un proceso se reincorpora por intercambio, el paginador adivina qué páginas usará. En vez de intercambiar todo el proceso, el paginador sólo trae a memoria las páginas necesarias. Así evita colocar en la memoria páginas que no se utilizarán, reduciendo el tiempo de intercambio y la cantidad de memoria física necesaria.

Este esquema requiere apoyo del hardware. Generalmente se añade un bit más a cada entrada de la tabla de páginas: un bit *válido* (indica que la página asociada se encuentra en memoria) – *inválido* (indica que está en disco). La entrada de páginas para una página que se ha traído se asigna de la manera habitual, pero para una página sin cargar únicamente se marca como "inválido".

Si se trata de acceder a una página que no se trajo a memoria, ocurrirá una trampa de *fallo de página*. El hardware de paginación, al traducir la dirección mediante la tabla de páginas, observará que el valor del bit es "inválido", generando una trampa para el sistema operativo. En esta situación la trampa es el resultado de la falla del **SO** al no

transferir a memoria una parte válida del proceso, por tanto, debemos corregir esta omisión. El procedimiento es bastante sencillo:

1. Consultamos una tabla interna (generalmente se conserva en el bloque de control del proceso) correspondiente al proceso para determinar si la referencia fue un acceso a memoria válido o inválido.
2. Si fue inválido, abortamos el proceso. Si se trató de una referencia válida, pero aún no hemos traído la página, la incorporamos.
3. Encontramos un marco libre.
4. Planificamos una operación para leer de disco la página deseada en el marco recién asignado.
5. Cuando ha concluido la lectura del disco, modificamos la tabla interna que se conserva junto con el proceso y la tabla de páginas para indicar que ahora la página se encuentra en memoria.
6. Reiniciamos la instrucción interrumpida por la trampa de dirección ilegal. El proceso ahora puede acceder a la página como si siempre se hubiera encontrado en memoria.

Es importante observar que, como almacenamos el estado del proceso interrumpido al ocurrir una falla de página, podemos reanudar el proceso *exactamente* en el mismo punto y estado, excepto que ahora la página deseada se encuentra en memoria y puede acceder a ella.

Los programas suelen poseer una localidad de referencias, lo que brinda un rendimiento aceptable en la paginación por demanda.

Hardware para apoyar la paginación por demanda:

- **Tabla de páginas**: esta tabla tiene la capacidad para marcar una entrada como inválida usando un bit válido-inválido o un valor especial de los bits de protección.
- **Memoria secundaria**: esta memoria contiene las páginas que no se conservan en memoria principal. La sección del disco que se usa para este fin se denomina *espacio de intercambios*.

Hay que imponer algunas otras restricciones arquitectónicas. Un aspecto crítico es la capacidad para reiniciar cualquier instrucción después de una falla de página. En la mayoría de los casos, una falla de página puede ocurrir en cualquier referencia a memoria. Si la falla ocurre al buscar la instrucción, podemos reiniciar efectuando de nuevo la búsqueda. Si ocurre al buscar un operando, debemos buscar de nuevo la instrucción decodificarla y luego buscar el operando.

Considere la instrucción MVC (mover carácter) del computador IBM System 360/370, que puede mover hasta 256 bytes de una localidad a otra (que pueden coincidir parcialmente). Si alguno de los bloques (fuente o destino) sobrepasa un límite de página, puede ocurrir una falla de página después de haber efectuado parte de la transferencia. Además, si los bloques fuente o destino se superponen es probable que se modifique el bloque fuente, por lo que no podríamos reiniciar la instrucción.

Este problema se resuelve de dos maneras:

1. El microcódigo calcula y trata de acceder a ambos extremos de los dos bloques. Si va a ocurrir una falla de página, sucederá en esta etapa, antes de modificar algo.
2. La otra solución utiliza registros temporales para contener los valores de las localidades sobrescritas.

Un problema arquitectónico similar se presenta en máquina que utilizan modos de direccionamiento especiales, incluyendo modos de autoincremento o autodecremento. Estos modos de direccionamiento utilizan un registro como apuntador e incrementan o reducen automáticamente el registro según se indique. El autodecremento reduce automáticamente el registro *antes* de usar su contenido como dirección del operando; el autoincremento aumenta automáticamente el registro *después* de usar su contenido como dirección del operando.

Ahora considere qué sucederá si provocamos una falla al tratar de almacenar en la localidad indicada por el registro que se autodecrementa. Para reiniciar la instrucción, debemos reasignar los registros con los valores que tenían antes de iniciar la ejecución de la instrucción. Una solución es crear un nuevo registro especial de estado para anotar el número de registro y la cantidad modificada para cualquier registro que cambie durante la ejecución de la instrucción. Este registro permitirá "deshacer" el efecto de una instrucción ejecutada parcialmente que provoque un fallo de página.

En un sistema de computación, la paginación se inserta entre la CPU y la memoria, y debe ser completamente transparente para el proceso de usuario.

8.3 Rendimiento de la paginación por demanda

La paginación por demanda puede tener un efecto considerable en el rendimiento del sistema de computación.

Una falla de página desencadena la siguiente situación:

1. Trampa para el **SO**.
2. Guardar los registros de usuario y el estado del proceso.
3. Determinar si la interrupción fue un fallo de página.
4. Verificar que la referencia a la página haya sido legal y determinar la ubicación de la página en el disco.
5. Leer de disco a marco libre
    a. Esperar en la cola del dispositivo hasta que se atienda la solicitud de lectura.
    b. Esperar durante el tiempo de posicionamiento o de latencia (o ambos) del dispositivo.
    c. Comenzar la transferencia de la página al marco libre.
6. Durante la espera, asignar la CPU a otro usuario (opcional).
7. Interrupción de disco (fin de E/S).
8. Guardar los registro y el estado del proceso de otro usuario.
9. Determinar si la interrupción provino del disco.
10. Corregir la tabla de páginas y las demás tablas para indicar que la página deseada se encuentra en memoria.
11. Esperar que la CPU se asigne nuevamente a este proceso.
12. Restablecer los registros de usuario, estado del proceso y la nueva tabla de páginas, y reanudar la instrucción interrumpida.

En cualquier caso, nos enfrentamos a tres componentes principales del tiempo de servicio de la falla de página:

- Servir la interrupción de falla de página.
- Leer la página.
- Reanudar el proceso.

La primera y tercera tarea pueden reducirse, con una codificación cuidadosa, a varios cientos de instrucciones. Estas tareas pueden constar de 1 a 100 microsegundos cada una. El tiempo de cambio de página, será cercano a los 24 milisegundos. El tiempo total de paginación será cercano a los 25 milisegundos, incluyendo hardware y software. Si una cola de procesos espera al dispositivo, tenemos que añadir el tiempo de cola.

Si tomamos un tiempo promedio de servicio de falla de página de 25 milisegundos y un tiempo de acceso a memoria de 100 nanosegundos, entonces el tiempo de acceso efectivo, expresado en nanosegundo es: 100+24.999.900 x p.

Vemos entonces que el tiempo de acceso efectivo es directamente proporcional a la tasa de falla de página.

Otro aspecto de la paginación por demanda es el manejo y uso global del espacio de intercambios. La E/S de disco para el espacio de intercambios generalmente es más rápida que con el sistema de archivos. Es más rápida porque el espacio de intercambios se asigna en bloques mucho mayores y no se usan las búsquedas de archivos ni los métodos de asignación indirecta. Por tanto, es posible que el sistema obtenga una mejor productividad de la paginación copiando todo el contenido del archivo al espacio de intercambios al iniciar el proceso, y luego efectuar la paginación por demanda desde el

espacio de intercambios. Otra opción más consistente es solicitar inicialmente páginas del sistema de archivos, pero escribirlas en el espacio de intercambio conforme se reemplazan. Esta estrategia asegurará que sólo se lean las páginas necesarias del sistema de archivos, pero que todos los intercambios de paginación posteriores se lleven a cabo con el espacio de intercambios. Este método parece ser el óptimo.

8.4 Reemplazo de páginas

Considere que, si un proceso de 10 páginas sólo emplea la mitad de ellas, entonces la paginación por demanda ahorra la E/S necesaria para cargar cinco páginas que nunca se usarán. Podemos aumentar el nivel de multiprogramación ejecutando el doble de procesos.

Si aumentamos nuestro nivel de multiprogramación, estamos *sobreasignando*. La sobreasignación se presentará de la siguiente manera: mientras se ejecuta un proceso de usuario, ocurre una falla de página. El hardware genera una trampa para el **SO**, el cual consulta sus tablas internas para ver si se trata de una falla de página o un acceso ilegal a la memoria. El sistema operativo determina en qué lugar del disco reside la página deseada, pero luego descubre que no quedan marcos libres en la lista; se está usando toda la memoria.

Al llegar a este punto, el sistema operativo tiene varias opciones, como abortar el proceso de usuario. Sin embargo, la paginación por demanda es algo que el sistema operativo hace para mejorar la utilización y la productividad del sistema de computación.

Podemos descargar un proceso liberando todos sus marcos y reduciendo el nivel de multiprogramación. Pero también se puede *reemplazar páginas*.

El reemplazo de páginas adopta la estrategia siguiente: si no hay ningún marco libre, encontramos un que no se esté utilizando en ese momento y lo liberamos. Podemos liberar un marco escribiendo en disco todo su contenido y modificando la tabla de páginas (y todas las demás tablas) para indicar que la página ya no se encuentra en memoria. Ahora se modifica la rutina de servicio de falla de página para incluir el reemplazo.

Observe que, si no quedan marcos libres, se requieren *dos* transferencias de páginas, duplicando el tiempo de servicios de la falla de página y aumentando el tiempo de acceso efectivo.

Este tiempo de procesamiento adicional se puede reducir utilizando un *bit* de modificación. Cada página o marco puede tener asociado un bit de modificación en el hardware. El hardware pone a uno el bit de modificación para una página cuando en ella se escribe una palabra o un byte, indicando que ha sido modificada. Cuando seleccionamos una página para reemplazo, examinamos su bit de modificación. Si el bit está activo, sabemos que la página fue modificada desde que se leyó de disco. En este caso, debemos escribir la página en disco. Si el bit de modificación no está activo, la página *no* ha sido modificada desde que se leyó en memoria. Por tanto, podemos evitar la escritura en disco de esta página, pues ya se encuentra allí. Esta técnica también se aplica a páginas de sólo lectura.

El reemplazo de páginas es esencial para la paginación por demanda, pues completa la separación entre la memoria lógica y la física.

8.5 Algoritmos de reemplazo de páginas

En general, queremos que presente la menos *tasa de fallas de páginas*.

Evaluamos un algoritmo ejecutándolo para una serie determinada de referencias a memoria y calculando el número de fallas de página. Podemos generar artificialmente las series de referencias o rastreando un sistema y anotando la dirección de cada referencia a memoria. En este último caso, consideramos sólo el número de página. Para determinar el número de fallas de páginas, necesitamos conocer también el número de marcos de página disponibles. Evidentemente, más marcos, menos fallas.

Una mala elección en el reemplaza aumenta la tasa de fallas de página y frena la ejecución de procesos, pero no provoca una ejecución incorrecta.

8.5.1 Algoritmo FIFO

El más sencillo (como siempre). Este algoritmo asocia a cada página el instante en el cual se trajo a memoria. Cuando hay que reemplazar una página, se elige la más antigua. Reemplazamos la página al inicio de la cola, y cuando se introduce en memoria una página, la insertamos al final de la cola.

Anomalía de Belady: suele suceder que el número de fallas para *n* páginas sea mayor que el número de fallas para *n-m* páginas (siendo $m > 0$). Para algunos algoritmos de reemplazo de páginas, la tasa de fallas de página puede *aumentar* al incrementarse el número de marcos asignados.

8.5.2 Algoritmo óptimo

Un algoritmo de reemplazo de páginas óptimo tiene la menor tasa de fallas de páginas de todos los algoritmos: un algoritmo óptimo, nunca presentará la anomalía de Belady. Este algoritmo consiste en:

Reemplazar la página que no se usará durante el mayor tiempo.

Por desgracia, es difícil implantar el algoritmo de reemplazo de páginas óptimo, ya que se requiere de un conocimiento futuro de la serie de referencias. Como resultado de esto, el algoritmo óptimo se utiliza sobre todo para estudios comparativos.

Este algoritmo no padece de la anomalía de Belady.

8.5.3 Algoritmo LRU

Si utilizamos el pasado reciente como aproximación de lo que sucederá en el futuro cercano, reemplazaremos la página *que no se ha utilizado* durante el mayor período de tiempo (LRU).

El reemplazo LRU asocia a cada página el instante en que se usó por última vez. LRU selecciona la página que no ha sido utilizada durante el mayor período de tiempo. Éste es el algoritmo óptimo de reemplazo de páginas que ve hacia atrás en el tiempo, no hacia adelante.

Este algoritmo se considera como bastante bueno. El problema principal es *cómo* implantar el reemplazo LRU. Este algoritmo puede requerir una ayuda considerable del hardware.

Son factibles dos implantaciones:

- **Contadores** (el más sencillo): a cada entrada de la tabla de páginas asociamos un registro de instante de uso y añadimos a la CPU un reloj lógico o contador. El reloj se incrementa con cada referencia a memoria. Siempre que se efectúa una referencia a una página, el contenido del registro del reloj se copia al registro de instante de uso en la tabla de páginas para esa página. Este esquema requiere una búsqueda en la tabla de páginas para encontrar la página menos recientemente usada. También se deben mantener los tiempos cuando se cambian las tablas de páginas.
- **Pila**: Se mantiene una *pila* de números de página. Cuando se hace referencia a una página, se saca de la pila y se coloca en la parte superior. De esta manera, en la parte superior de la pila siempre se encuentra la página más recientemente usada, y en la parte inferior, la menos recientemente usada. Puesto que las entradas deben sacarse de la mitad de la pila, puede implantarse mejor con una lista doblemente ligada, con apuntadores al inicio y al final. Al sacar una página y colocarla en la cima de la pila, a lo sumo hay que cambiar seis apuntadores. Cada actualización es un poco más costosa, pero no hay que buscar para reemplazar

Este algoritmo no padece de la anomalía de Belady.

Algoritmo de pila: es un algoritmo para el cual se puede demostrar que el conjunto de páginas en memoria para *n* marcos es siempre un *subconjunto* del conjunto de páginas que estarían en memoria con *n+1* marcos.

La implantación de LRU no podría concebirse sin ayuda del hardware. La actualización de los registros de reloj o la pila debe efectuarse para *cada* referencia a memoria.

8.5.4 Algoritmos aproximados al LRU

Algunos sistemas no ofrecen ningún apoyo del hardware, por lo que hay que emplear otros algoritmos. Sin embargo, muchos sistemas ofrecen cierta ayuda, en la forma de *bit de referencia*. El hardware coloca a uno el bit de referencia para una página cada vez que se hace una referencia a ella. Los bits de referencia están asociados a cada entrada de la tabla de páginas.

En un principio, el **SO** borra todos los bits (asignándoles 0). Al ejecutarse un proceso de usuario, el hardware activa (asignando 1) el bit asociado a cada página referida. Después de cierto tiempo, examinando los bits de referencia podemos determinar cuáles páginas se han utilizado y cuáles no, aunque no sabemos el *orden* de uso. Esta información parcial de la ordenación nos lleva a varios algoritmos de reemplazo de páginas que se aproximan al LRU.

Algoritmo de bits adicionales de referencia

Podemos obtener información adicional de la ordenación anotando los bits de referencia a intervalos regulares. En una tabla de memoria podemos conservar un byte para cada página. A intervalos regulares, una interrupción del cronómetro transfiere el control al **SO**. Éste desplaza el bit de referencia de cada página al bit de orden superior de su byte de ocho bits, desplazando los otros bits una posición a la derecha y descartando el bit de orden inferior. Estos registros de desplazamiento de ocho bits contienen la historia de la utilización de la página durante los últimos ocho periodos. Por supuesto, el número de bits históricos puede variar. En el caso extremo, el número puede reducirse a cero, dejando únicamente el bit de referencia. Esta versión se denomina algoritmo de reemplazo de páginas de *segunda oportunidad*.

Algoritmo de segunda oportunidad

El algoritmo básico para el reemplazo de segunda oportunidad es un algoritmo FIFO. Sin embargo, cuando se selecciona una página, examinaremos su bit de referencia, si es igual a 0, reemplazamos la página, si es 1, damos a la página una segunda oportunidad y pasamos a seleccionar la siguiente página en el orden FIFO. Cuando a una página se le brinda una segunda oportunidad, se borra su bit de referencia y se establece como su instante de llegada el momento actual.

Una forma de implantar el algoritmo de segunda oportunidad consiste en usar una cola circular. Un apuntador indica cuál es la siguiente página que se reemplazará. Cuando se requiere un marco, el apuntador avanza hasta encontrar una página con bit de referencia a 0; conforme avanza, borra los bits de referencia.

Si todos los bits están activos, el reemplazo de segunda oportunidad se convierte en reemplazo FIFO.

Algoritmo LFU

Un algoritmo de reemplazo *menos frecuentemente usada* (LFU) mantiene un contador del número de referencias que se han hecho para cada página. Se reemplaza la página con el menor recuento. Este algoritmo tiene problemas cuando una página se usa mucho en la fase inicial de un proceso, pero después ya no se utiliza. Una solución consiste en desplazar los recuentos un bit a la derecha a intervalos regulares, formando un recuento promedio de utilización que disminuye exponencialmente.

Algoritmo MFU

Reemplazo *más frecuentemente usado* que se basa en el argumento de que la página con el menor recuento probablemente acaba de llegar y aún tiene que usarse.

Algoritmos adicionales

Si consideramos tanto el bit de referencia como el bit de modificación como un par ordenado,

- (0,0) ni usada ni modificada.
- (0,1) no usada (recientemente), pero modificada.
- (1,0) usada pero no modificada.
- (1,1) usada y modificada.

Reemplazamos la página en la clase inferior no vacía. Si hay varias páginas, utilizamos reemplazo FIFO o elegimos al azar.

8.5.5 Algoritmos ad hoc

Con frecuencia se utilizan otros algoritmos junto a un algoritmo específico de reemplazo de página. Por ejemplo, es habitual que los sistemas mantengan un *depósito* de marcos vacíos. Cuando ocurre una falla de página se selecciona un marco víctima de la manera acostumbrada. Sin embargo, la página deseada se lee en un marco libre del depósito antes de sacar a la víctima. Este método permite que el proceso se reanude lo más pronto posible, sin tener que esperar que salga la página víctima. Cuando posteriormente se saque la víctima, su marco se añadirá al depósito de marcos libres.

Una extensión de este concepto es mantener una lista de páginas modificadas. Cuando el dispositivo de paginación esté inactivo, se selecciona una página modificada y se escribe en disco; luego se pone a cero su bit de modificación. Esta estrategia aumenta la probabilidad de que una página esté limpia cuando se selecciona para reemplazo.

Otra modificación es mantener un depósito de marcos libres pero recordando qué página se encuentra en cada uno. Como el contenido de los marcos no se modifica al escribirlos en disco, se puede reutilizar la página antigua directamente del depósito de marcos libres si se necesita antes de volver a ocupar el marco. En este caso no se requiere ninguna E/S.

8.6 Asignación de marcos

El caso más sencillo de memoria virtual es el sistema monousuario.

Teniendo una lista de marcos libres, se van asignando hasta agotarse. Una vez agotados los marcos libres, se empleará un algoritmo de reemplazo de páginas para seleccionar una de de las páginas ocupadas. Al terminar el proceso se colocan todos los marcos en la lista de marcos libres.

Existen diferentes variantes para esta sencilla estrategia. Podemos requerir que el **SO** asigne todo su espacio de almacenamiento temporal y de tablas de la lista de marcos libres. Cuando el **SO** no usa este espacio, puede aprovecharse para la paginación de los usuarios. Podemos tratar de reservar todo el tiempo tres marcos libres en la lista de marcos libres. Así si se presenta una falla de página, hay un marco libre para introducir la página. Mientras se lleva a cabo el intercambio de la página, puede seleccionarse un reemplazo, que se escribe a disco mientras continúa la ejecución del proceso de usuario.

La estrategia básica siempre es: al proceso usuario se lo asigna a cualquier marco libre.

8.6.1 Número mínimo de marcos

Existen varias restricciones para nuestras estrategias de asignación de marcos. No podemos asignar más del total de marcos libres (a menos que se compartan páginas). También hay un número mínimo de marcos que deben asignarse. Este número está definido por la arquitectura del conjunto de instrucciones. Recuerde que cuando ocurre una falla de página antes de terminar la ejecución de una instrucción, ésta se debe reiniciar. Por consiguiente, debemos contar con marcos suficientes para todas las páginas a las cuales pueda hacer referencia una instrucción.

En teoría, una sencilla instrucción de carga puede hacer referencia a una dirección indirecta que puede hacer referencia a una dirección indirecta (en otra página), y así, hasta alcanzar cada una de las páginas en memoria virtual. Por ende, en el peor de los casos, toda la memoria virtual debe estar en memoria física. Para superar este obstáculo debemos poner un límite en los niveles de indirección. Esto se logra usando un contador que se decremente con cada indirección, hasta llegar a 0, generando una trampa para el **SO**.

El número máximo de marcos está definido por la cantidad de memoria física disponible.

8.6.2 Algoritmos de asignación

La manera más fácil de dividir m marcos entre n procesos consiste en otorgar a cada uno una parte igual m/n marcos. Este esquema se llama *asignación equitativa*.

Una alternativa es reconocer que los diversos procesos necesitarán cantidades distintas de memoria.

Para resolver este problema podemos usar la asignación proporcional. Asignamos la memoria disponible a cada proceso de acuerdo al tamaño de éste. De esta manera, ambos procesos comparten los marcos disponibles de acuerdo con sus "necesidades" (*Asignación proporcional*).

Por supuesto, tanto en la asignación equitativa como en la proporcional, la asignación para cada proceso puede variar de acuerdo con el nivel de multiprogramación. Si aumenta el nivel de multiprogramación, cada proceso perderá algunos marcos para proporcionar al nuevo proceso la memoria necesaria. Por otra parte, si disminuye el nivel de multiprogramación, los marcos asignados al proceso que sale pueden distribuirse entre los restantes.

Observe que con la asignación proporcional o la equitativa, se trata de igual manera a un proceso de alta prioridad y a uno de baja prioridad. Sin embargo, por definición, queremos darle más memoria al proceso de alta prioridad para acelerar su ejecución, en perjuicio de los de baja prioridad.

Una estrategia consiste en usar un esquema de asignación proporcional donde la tasa de marcos no dependa de los tamaños relativos de los procesos, sino de sus prioridades o de una combinación de tamaño y prioridad.

Otra estrategia consiste en permitir que un proceso de alta prioridad selecciones para su reemplazo los marcos de un proceso de menor prioridad.

8.7 Hiperpaginación

Si el número de marcos asignados a un proceso de baja prioridad desciende por debajo del número mínimo requerido por la arquitectura del computador, debemos suspender la ejecución de ese proceso. Luego debemos descargar sus páginas restantes, liberando todos los marcos asignados. Esta medida introduce un nivel de intercambios de la planificación a mediano plazo de CPU.

Observe cualquier proceso que no tiene marcos "suficientes". Aunque técnicamente es posible reducir al mínimo el número de marcos asignados, hay un número (mayor) de páginas que se usan activamente. Si el proceso no tiene este número de marcos, provocará fallas de página muy frecuentemente; en este momento debe reemplazar alguna página. Sin embargo, como todas sus páginas están activas, debe reemplazar una página que casi de inmediato se volverá a necesitar. Por consiguiente, pronto vuelve a fallar, y una y otra vez más. El proceso sigue generando fallas, reemplazando páginas por las cuales fallará y se reincorporará enseguida.

A esta altísima actividad de paginación se le llama *hiperpaginación*. Un proceso está en hiperpaginación si emplea más tiempo paginando que ejecutando.

8.7.1 Causas de la hiperpaginación

La hiperpaginación ocasiona severos problemas de rendimiento.

Al aumentar el nivel de multiprogramación, también aumenta la utilización de la CPU, aunque cada vez con mayor lentitud hasta alcanzar un punto máximo. Si se incrementa más el nivel de multiprogramación, comienza la hiperpaginación y la utilización de la CPU decae rápidamente. Al llegar a este punto, para aumentar la utilización de la CPU y detener la hiperpaginación debemos *reducir* el nivel de multiprogramación.

Los efectos de hiperpaginación se pueden limitar utilizando un *algoritmo de reemplazo local* (o por prioridades). Con el reemplazo local, si en un proceso comienza la hiperpaginación, no puede robar marcos a otro proceso y provocar que éste también entre en hiperpaginación. Sin embargo, si los procesos están en hiperpaginación, la mayor parte del tiempo pueden encontrarse en la cola del dispositivo de paginación. El tiempo promedio de servicio de una falla de página aumentará, por ser mayor el tamaño de promedio de la cola del dispositivo de paginación. Por consiguiente, el tiempo de acceso efectivo aumentará incluso para un proceso que no esté en hiperpaginación.

Para evitar la hiperpaginación debemos ofrecer a un proceso todos los marcos que necesita. ¿Cómo sabemos cuántos "necesita"? Para saberlo existen varias técnicas: la técnica del área activa comienza buscando cuántos marcos está usando realmente un proceso. Esta estrategia define el *modelo de localidad* de la ejecución de procesos.

El modelo de localidad establece que un proceso, durante su ejecución, pasa de una localidad a otra.

Localidad: conjunto de páginas que se utilizan conjuntamente.

Un problema generalmente está compuesto por varias localidades distintas, las cuales pueden superponerse. Al llamar a una subrutina se define una nueva localidad. En ésta se efectúan referencias a memoria de las instrucciones de la subrutina, sus variables locales y un subconjunto de las variables globales. Al salir de la subrutina, el proceso abandona esta localidad, pues ya no se usan las variables locales e instrucciones de la subrutina. Pero más adelante podemos regresar a esta localidad. Así, vemos que las localidades están definidas por la estructura del programa y sus estructuras de datos. El modelo de localidad establece que todos los programas presentarán esta estructura básica de referencias a memoria.

Suponga que a un proceso le asignamos suficientes marcos para colocar su localidad actual. Generará fallas por las páginas en su localidad hasta que todas se encuentren en memoria; luego no fallará hasta que cambie de localidad.

8.7.2 Modelo del área activa

El *modelo del área activa* (*conjunto de páginas de mayor demanda*) se basa en la suposición de la localidad. El modelo utiliza un parámetro para definir la *ventana* del área activa. La idea es examinar las referencias más recientes a páginas. El conjunto de páginas en las referencias más recientes constituye el *área activa*. Si una página está en uso pertenecerá al área activa, y si ya no se usa se descartará del área activa tras unidades de tiempo después de su última referencia. De esta manera, el área activa es una aproximación de la localidad del programa.

La exactitud del área activa depende de la acción de Si es demasiado pequeño no abarcará toda el área activa; si es demasiado grande, puede abarcar varias localidades.

La propiedad más importante de las áreas activas es su tamaño. **â**

D=TAA

Si la demanda total es mayor que el número total de marcos disponibles se producirá la hiperpaginación.

El **SO** supervisa el área activa de cada proceso y le asigna marcos suficientes para proporcionarle el tamaño del área activa. Si hay suficientes marcos adicionales, se puede iniciar otro proceso. Si aumenta la suma de los tamaño de las áreas activas, excediendo el número total de marcos disponibles, el **SO** selecciona un proceso y lo suspende.

La estrategia del área activa evita la hiperpaginación manteniendo a la vez el nivel de multiprogramación lo más alto posible.

Con el método del área activa el problemas es el seguimiento de esa área activa. Podemos aproximarnos al modelo del área activa mediante interrupciones a intervalos fijos de un cronómetro y un bit de referencia.

8.7.3 Frecuencia de fallas de página

El modelo del área activa tiene bastante éxito, y el conocimiento del área activa puede resultar útil para la prepaginación, pero parece ser una manera bastante torpe de controlar la hiperpaginación. La estrategia de *frecuencia de falla de página* sigue una ruta más directa.

La hiperpaginación presenta una elevada tasa de fallas de página, por lo que queremos controlar dicha tasa. Cuando es demasiado alta sabemos que un proceso necesita más marcos, y si es demasiado baja, puede ser que el proceso tenga demasiados marcos. Podemos establecer límites superior e inferior para la tasa deseada de fallas de página. Si la tasa de fallas de página excede el límite superior, asignamos otro marco a ese proceso; si la tasa cae por debajo del límite inferior, quitamos un marco al proceso. Así podemos medir y controlar directamente la tasa de fallas de página para evitar la hiperpaginación.

Al igual que sucede en la estrategia del área activa, posiblemente tengamos que suspender un proceso.

8.8 Otras consideraciones

8.8.1 Asignación global frente a asignación local

Cuando varis procesos compiten por marcos, podemos clasificar los algoritmos de reemplazo de páginas en dos grandes categorías: *reemplazo global* y *local*. El global permite que un proceso seleccione un marco para reemplazar de entre todo el conjunto de marcos, incluso si ese marco está asignado a otro proceso; un proceso puede quitar un marco a otro. El reemplazo local requiere que cada proceso selecciones sólo de su propio conjunto de marcos asignados.

Un problema del algoritmo de reemplazo global es que un proceso no puede controlar su propia tasa de fallas de página. En local, la tasa sólo se ve afectada por su propio comportamiento.

El reemplazo global generalmente brinda mayor productividad.

8.8.2 Prepaginación

La *prepaginación* es un intento de evitar que un proceso produzca fallas de página por cada página que necesita, cuando se lo reanuda. La estrategia consiste en traer a memoria al mismo tiempo todas las páginas que se necesitarán.

En un sistema que utiliza el modelo del área activa, por ejemplo, con cada proceso debemos conservar una lista de las páginas que hay en su área activa. Si tenemos que suspender un proceso recordamos cuál era el área activa para el proceso. Cuando el proceso se reanuda, automáticamente incorporamos toda el área activa antes de reanudar el proceso.

La prepaginación puede ser una ventaja en algunos casos. La cuestión es sencillamente si el costo de la prepaginación es menor al costo del servicio de las fallas de página correspondientes. Puede ser que ya no se usen muchas de las páginas devueltas a memoria por la prepaginación.

8.8.3 Tamaño de página

Los diseñadores de un **SO** para una máquina existente pocas veces tienen opciones respecto al tamaño de página (potencias de 2).

¿Cómo seleccionamos el tamaño de página?. Un factor es el tamaño de la tabla de páginas. Para un espacio de memoria virtual establecido, al reducir el tamaño aumenta el número de páginas y, por ende, el tamaño de la tabla de páginas. Como cada proceso activo debe tener su propia copia de la tala de páginas, vemos que es deseable un tamaño de página grande.

Por otra parte, la memoria se utiliza mejor con páginas más pequeñas. Para minimizar la fragmentación interna necesitamos un tamaño de página pequeño.

Otro problema es el tiempo necesario para leer o escribir una página. El tiempo de E/S está compuesto por tiempo de posicionamiento, latencia y transferencia. El tiempo de transferencia es proporcional a la cantidad transferida (o sea, el tamaño de página), hecho que aparentemente apunta a favor de un tamaño de página pequeño. Sin embargo, recuerde que los tiempos de posicionamiento y latencia normalmente convierten el tiempo de transferencia en insignificante.

Sin embargo, con un menor tamaño de página debe reducirse el número total de Operaciones de E/S, ya que la localidad mejorará. Un tamaño de página menor permite que cada página se ajuste con mayor precisión a la localidad del programa. Con un tamaño de página menor obtenemos una mejor *resolución*, lo que nos permite aislar la memoria que realmente necesitamos. Si el tamaño de página es mayor tenemos que transferir y asignar no sólo lo que se ejecuta, sino además todo lo que se encuentre en la página, se requiera o no.

Un proceso de 200K que sólo utiliza la mitad de esa memoria genera una sola falla de página si el tamaño de página es de 200K, pero generaría 102.400 fallas de página si el tamaño fuera un byte. Para minimizar el número de fallas de páginas necesitamos un tamaño de página mayor.

8.8.4 Estructura de los programas

La paginación por demanda está diseñada para ser transparente para el programa de usuario, aunque el rendimiento del sistema puede mejorarse si se es consciente de la paginación por demanda subyacente.

Una cuidadosa selección de las estructuras de datos y programación puede aumentar la localidad y, por consiguiente, disminuir la tasa de fallas de página y el número de páginas en el área activa. Una pila tiene buena localidad ya que el acceso siempre es a la parte superior.

Otros factores de peso incluyen la velocidad de búsqueda, el número total de referencias a memoria y el total de páginas a las que se accedió.

En una etapa posterior, el compilador y el cargador pueden tener un efecto considerable sobre la paginación. Al separar código y datos y generar código reentrante, las páginas de código pueden ser de sólo lectura y, por tanto, nunca tendrán que modificarse. Las páginas limpias no tienen que sacarse y escribirse cuando se reemplazan. El cargador puede evitar la colocación de rutinas que crucen los límites de página, manteniendo toda una rutina dentro de la misma página. Muchas veces pueden agruparse en la misma página las rutinas que se llaman entre sí.

8.8.5 Fijación de páginas para E/S

Cuando se utiliza la paginación por demanda, en ocasiones tenemos que permitir que algunas páginas se *fijen* en memoria.

Debemos asegurar que no se presente la siguiente secuencia de sucesos: Un proceso emite una solicitud de E/S y se coloca en la cola para ese dispositivo de E/S. Mientras tanto, se asigna la CPU a otros procesos. Estos generan fallas de página y, usando un algoritmo de reemplazo global, uno de ellos reemplaza la página que contiene el buffer de memoria del proceso en espera. Se sacan las páginas y más tarde, cuando la solicitud de E/S avanza hasta el inicio de la cola del dispositivo, se efectúa la E/S para la dirección especificada. Sin embargo, este marco se está utilizando ahora para una página que pertenece a otro proceso.

Hay dos soluciones para esto: una consiste en no ejecutar nunca la E/S a memoria de usuario, sino copiar los datos entre la memoria del sistema y la de usuario. Para escribir un bloque en cinta, primero copiamos el bloque a la memoria del sistema y luego lo escribimos en cinta.

Esta copia adicional puede generar un tiempo de procesamiento añadido inaceptable. Otra solución consiste en permitir que las páginas se *fijen* a memoria. A cada marco se asocia un *bit de fijación*, si se ha fijado el marco, no puede seleccionarse para reemplazo. En este esquema, para escribir un bloque en cinta, fijamos en memoria las páginas que contienen el bloque.

Otra aplicación del bit de fijación se refiere al reemplazo normal de páginas. Considere la siguiente secuencia de sucesos. Un proceso de baja prioridad genera una falla. Al seleccionar un marco para su reemplazo, el sistema de paginación lee en memoria la página requerida. Listo para continuar, el proceso de baja prioridad entra en la cola de procesos listos y espera la CPU. Al ser de baja prioridad, puede que el planificador tarde en asignarle la CPU. Mientras el proceso de baja prioridad espera, uno de alta genera una falla. Buscando un reemplazo, el sistema de paginación encuentra una página que esté en memoria pero que no ha sido modificada ni tiene referencias: la página que acaba de incorporar el proceso de baja prioridad, parece perfecta para el reemplazo, está limpia y no hay que escribirla al sacarla, y aparentemente no se ha utilizado en mucho tiempo.

La decisión sobre si el proceso de mayor prioridad puede reemplazar un marco del de menor prioridad tiene un carácter político. Después de todo, estamos demorando el proceso de baja prioridad para beneficiar el de alta. Por otra parte, estamos desperdiciando el esfuerzo aplicado para incorporar la página del proceso de menor prioridad. Si decidimos impedir el reemplazo de una página recién incorporada hasta que se utilice por lo menos una vez podemos usar el bit de fijación para implantar este mecanismo. Cuando se selecciona una página para su reemplazo, se activa el bit de fijación y permanece activo hasta que se despache el proceso que provocó la falla.

Sin embargo, puede ser peligro usar el bit de fijación si se activa y nunca desactiva. Se llegase a ocurrir esta situación, el marco fijado sería inutilizable.

8.8.6 Tabla de páginas invertida

Casi siempre hay una tabla de páginas asociada a cada proceso. La tabla de páginas contiene una entrada por cada página virtual que usa el proceso. Ésta es una representación natural de la tabla ya que los procesos hacen referencia a las páginas por medio de direcciones virtuales de las páginas. Los espacios de direcciones de memoria virtual pueden ser del orden de gigabytes, generando tablas con millones de entradas. Estas tablas pueden consumir grandes cantidades de memoria física, sólo para controlar la forma en que se utiliza el resto de la memoria.

Para resolver este problema, algunos **SO** utilizan *tablas de páginas invertidas*. Esta tabla tiene una entrada para cada página real de memoria. Luego con cada entrada se asocia la dirección virtual de la página almacenada en esa localidad de memoria real, de manera que sólo hay una tabla de páginas en todo el sistema y sólo tiene una entrada para cada página de memoria física.

Cada dirección virtual consiste en una tripleta:

<idProceso, número-página, desplazamiento>

Cuando ocurre una referencia a memoria, se presenta al subsistema de memoria una parte de la dirección virtual, consistente en <idProceso, número-página>. Luego se recorre la tabla de páginas invertida para buscar una coincidencia. Si se encuentra un valor igual, digamos en la entrada *n*, entonces se genera la dirección física <*n*-desplazamiento>. Si no hay coincidencia, entonces ocurre una falla de página, indicando que se ha generado una dirección ilegal o que la página en cuestión no se encuentra en memoria.

Al conservar información sobre qué página de memoria virtual se almacena en cada marco físico, estas tablas reducen su tamaño. Pero no tiene información si una página virtual no está actualmente en memoria. Para esto hay que mantener tablas de páginas externas. Éstas se parecen a las tablas de páginas tradicionales para cada proceso, que contienen información sobre la localización de cada página virtual. Como sólo se hace referencia a estas tablas cuando ocurre una falla de página, no es necesario que estén disponibles inmediatamente. Por desgracia, una falla de página puede provocar ahora que el manejador de la memoria virtual genere otra falla de página para traer la tabla de páginas externa que necesita con el propósito de localizar la página virtual en el almacenamiento auxialiar.

Este esquema aumenta el tiempo necesario para buscar en la tabla cuando hay una referencia a una página.

Como la tabla de páginas invertidas está ordenada por dirección física, pero las búsquedas se efectúan por direcciones virtuales, habría que recorrer toda la tabla para encontrar una coincidencia. Para aliviar este problema usamos una tabla de dispersión para limitar la búsqueda a una, o unas cuantas, de las entradas de la tabla. Por supuesto, cada acceso a la tabla de dispersión añade una referencia a memoria al procedimiento. Para mejorar el rendimiento usamos registros asociativos de memoria para contener las entradas localizadas recientemente.

También ocasiona problemas la implantación de memoria compartida en sistemas que emplean páginas invertidas. La memoria compartida se implanta habitualmente como dos direcciones virtuales que se transforman en una misma dirección física.

8.9 Segmentación por demanda

Para la paginación por demanda, se requiere una considerable cantidad de hardware para implantarla. Para sistemas menos eficientes, se usa la segmentación por demanda.

El OS/2 asigna la memoria por segmentos, en vez de páginas y los controla por medio de *descriptores de segmentos*, que incluyen información sobre el tamaño, protecciones y ubicación del segmento. Un proceso no requiere que todos sus segmentos se encuentren en memoria para poder ejecutarse. En vez de esto, el descriptor de segmento contiene un bit de validez para cada segmento, el cual indica si se encuentra actualmente en memoria. Cuando un proceso direcciona un segmento que contiene datos

o código, el hardware examina este bit de validez. Si el segmento está en memoria principal, el acceso continúa sin problemas; en caso contrario se genera una trampa al **SO** (falla de segmento). El OS/2 saca, en un intercambio, un segmento al almacenamiento secundario e incorpora todo el segmento requerido. Luego continúa la instrucción interrumpida.

Para determinar que segmento se reemplazará en caso de que ocurra una falla de segmento, el OS/2 utiliza otro bit, en el descriptor de segmento, llamado *bit de acceso*. Un bit de acceso se activa al escribir o leer cualquier byte en el segmento. Se mantiene una cola que contiene una entrada por cada segmento en memoria. Después de cada porción de tiempo, el **SO** coloca al inicio de la cola cualesquiera segmentos con el bit de acceso activado; luego pone a cero todos los bits de acceso. De esta manera, la cola se mantiene ordenada con los segmentos de utilización más reciente al inicio. Además, el OS/2 ofrece llamadas al sistema que los procesos pueden emplear para informar al **SO** sobre que segmentos pueden descartarse o permanecer siempre en la memoria. Esta información se utiliza para reorganizar las entradas en la cola. Cuando ocurre una trampa de segmento inválido, las rutinas de administración de memoria primero determinan si hay espacio suficiente en memoria para albergar al segmento. Puede efectuarse una compactación de la memoria para eliminar la fragmentación externa. Si, después de la compactación, aún no hay bastante memoria libre, se lleva a cabo el reemplazo del segmento. Para el reemplazo, se escoge el segmento del final de la cola y se escribe en el almacenamiento de intercambio. Si el espacio recién liberado tiene el tamaño suficiente para contener el segmento solicitado, entonces éste se lee en el segmento desocupado, se actualiza el descriptor de segmento, y se coloca el segmento al inicio de la cola. De lo contrario, se compacta la memoria y se repite el procedimiento.

Se debe resaltar que la segmentación por demanda requiere un considerable tiempo de procesamiento adicional, por lo que ésta no es un medio óptimo para aprovechar al máximo los recursos de un sistema de computación. Por consiguiente, la segmentación por demanda es un compromiso razonable de funcionalidad dadas las restricciones del hardware que imposibilitan la paginación por demanda.

9. Administración del Almacenamiento Secundario

9.1 Antecedentes

Sería una situación ideal si todos los programas y los datos que acceden residieran en forma permanente en memoria principal, pero esto no es posible.

9.2 Estructura del Disco

9.2.1 Estructura física

Cada disco tiene forma circular plana, con ambas superficies cubiertas con material magnético, la información se graba en la superficie. Cuando el sistema está en uso, el disco gira permanentemente a alta velocidad.

La superficie del disco se divide lógicamente en pistas, donde la información es grabada o leída por medio de cabezas de lectoescrituras. Un disco de cabeza fija tiene una cabeza para cada pista (carísimo). Lo habitual es utilizar un disco de cabeza móvil, que posee una sola cabeza que se desplaza hacia adentro y hacia fuera para acceder a las distintas pistas. Para obtener mayor capacidad de almacenamiento, se utiliza una cabeza por cada lado del disco, además de apilar varios de ellos donde cada uno tiene su propio par de cabezas de lectoescritura.

El hardware para un sistema de disco puede dividirse en dos partes:

1. Unidad de disco: la parte mecánica, incluyendo el motor del dispositivo, las cabezas y la lógica relacionada.
2. Controladora del disco: determina la interacción lógica con la computadora, recibiendo instrucciones del procesador y ordenando a la unidad de disco que ejecute la instrucción.

Esta división de tareas permite que varias unidades de disco se conecten al mismo controlador. Algunas poseen memoria caché.

Para hacer referencia a la información en disco se emplea una dirección compuesta por varias partes: unidad, superficie, pista y sector. Un sector es la menor unidad de información que puede leerse o escribirse en disco.

Para acceder a un sector, las cabezas deben moverse a la pista correcta (tiempo de posicionamiento) y esperar a que el sector solicitado pase por debajo de la cabeza (tiempo de latencia).

Si para direccionar un sector específico, se necesitan un número de pista (o de cilindro), un número de superficie y un número de sector, podemos considerar al disco como un arreglo tridimensional de sectores, el cual es tratado por el **SO** como un arreglo unidimensional de bloques del disco, donde cada bloque es un sector.

9.2.2 Directorio del dispositivo

En él se indican cuáles son los archivos que se encuentran en el disco. Es una lista de los nombres de los archivos e incluye información como la ubicación del archivo en el disco, su longitud, tipo, etc. Esta información se graba en una dirección fija del disco.

9.3 Administración del Espacio Libre

Como la cantidad de es limitada, es necesario reutilizar el espacio usado por los archivos eliminados. Para controlar el espacio disponible en disco, el sistema mantiene una lista de espacio libre, donde se registran todos los bloques del disco que no están asignados a un archivo. Para crear un archivo, recorremos la lista de espacio libre hasta encontrar espacio suficiente, y se lo asigna al nuevo archivo; luego, se elimina este espacio de la lista. Cuando un archivo es eliminado se agrega su espacio en disco a la lista de espacio libre.

9.3.1 Vector de bits

Es posible implementar la lista de espacio libre como un vector o mapa de bits, donde cada bloque se representa con un bit. Si es 0 el bloque está libre, sino es 1.

Este mecanismo es bastante sencillo y eficiente para encontrar *n* bloques libres consecutivos en el disco. No resulta eficiente en el caso de que para la mayoría de los accesos no se conserve todo el vector en memoria principal.

9.3.2 Lista ligada

Otra estrategia consiste en enlazar todos los bloques libres del disco, manteniendo un puntero al primer bloque libre, el cual contiene otro puntero al siguiente bloque libre, etc.

Este esquema no es eficiente, pues para recorrer la lista, tenemos que leer cada uno de los bloques, lo que representa un tiempo considerable de E/S.

9.3.3 Agrupamiento

Una variante de la lista ligada es almacenar en el primer bloque las direcciones de *n* bloques libres. Los primeros *n-1* bloques están libres y el último contiene la dirección en disco de otro bloque que contiene las direcciones de otros *n* bloques libres.

9.3.4 Recuento

Tenemos la dirección del primer bloque libre y los *n* bloques libres contiguos que lo siguen. Cada entrada de la lista de espacio libre consiste en una dirección en disco y un recuento.

9.4 Métodos de asignación

9.4.1 Asignación contigua

Este método requiere que cada archivo ocupe un conjunto de direcciones contiguas en el disco. Las direcciones en disco definen una ordenación lineal.

La entrada del directorio para cada archivo indica la dirección del bloque inicial y la longitud del área asignada al archivo.

Si el acceso es secuencial, el sistema de archivos tiene la dirección en disco del último bloque al que se hizo referencia, y cuando es necesario, accede al bloque siguiente.

El inconveniente consiste en encontrar espacio contiguo disponible para un nuevo archivo a partir de una lista de huecos libres, generalmente se utilizan las estrategias del primer ajuste, el mejor ajuste y el pero ajuste. Al asignar y eliminar archivos el espacio libre en disco se divide en pequeños pedazos, surge entonces la fragmentación externa.

Otro problema es el poder determinar cuánto espacio se necesita para un archivo que varíe con el tiempo. Si asignamos poco espacio, el archivo puede no tener posibilidad de extenderse. Para solucionar esto surgen dos posibilidades; una es abortar el programa de usuario, asignar más espacio y ejecutar nuevamente el programa. Su costo es alto, entonces el usuario asignará más espacio del necesario, con el desperdicio que ello representa.

La otra posibilidad es encontrar un hueco más grande, copiar el archivo al nuevo espacio y liberar el anterior.

9.4.2 Asignación enlazada

En este tipo de asignación, el archivo es una lista enlazada de bloques de disco que pueden hallarse en cualquier parte del mismo. El directorio contiene un puntero al primero y al último bloque del archivo. Cada bloque tiene un puntero al próximo bloque del archivo.

Es fácil crear un archivo: simplemente se crea una nueva entrada en el directorio del dispositivo. En un principio se asigna un valor inicial nulo (el valor fin de lista del puntero) para representar un archivo vacío, y también se asigna cero al campo de tamaño. Una escritura a un archivo quita el primer bloque disponible de la lista de espacio libre y escribe en él; luego se enlaza el nuevo bloque al final del archivo. Para leer un archivo, basta con leer los bloques siguiendo los punteros. No hay fragmentación externa.

El inconveniente con este tipo de asignación es que sólo puede aplicarse eficazmente al acceso secuencial. Cada acceso a un puntero implica una lectura al disco,

por lo que es poco eficiente para el acceso directo para archivos. Otro inconveniente es el espacio requerido por el puntero, que nos quita espacio para los datos.

El problema más serio es la confiabilidad. Un error en el software del **SO** o un problema en el hardware, puede provocar que se elija un puntero incorrecto, enlazando el archivo a la lista de espacio libre o a otro archivo. Una solución consiste en usar listas doblemente ligadas o almacenar en cada bloque el nombre del archivo y el número de bloque relativo, pero este esquema requiere aún más espacio para cada archivo.

Una variante de este tipo de asignación es utilizar una tabla de asignación de archivos (FAT), que tiene una entrada para cada bloque del disco y está indexada por número de bloque. El directorio contiene el número del primer bloque del archivo y la entrada obtenida con ese número guarda el número del siguiente bloque. Esta cadena continúa hasta el último bloque, para el cual existe una marca especial de fin de archivo en la entrada de la tabla.

9.4.3 Asignación indexada

Este tipo de asignación resuelve algunos problemas de la asignación enlazada, reuniendo todos los punteros en un solo lugar: el bloque de índices.

Cada archivo tiene su propio bloque de índices, el cual es un arreglo de direcciones de bloques en disco. El directorio contiene la dirección del bloque de índices.

Al crear el archivo, se asigna nulo a todos los apuntadores del bloque de índices. Ya que puede usarse cualquier bloque libre del disco, la asignación indexada permite el acceso directo sin el problema de la fragmentación externa.

Sin embargo, el desperdicio de espacio puede ser mayor que en la asignación enlazada.

Surge aquí una duda sobre el tamaño de los bloques de índices. Surgen estos mecanismos:

- *Esquema enlazado*: se enlazan varios bloques de índices. Un bloque de índices puede contener el nombre del archivo y las primeras *n* direcciones de bloques en el disco, la siguiente dirección (la última palabra en el bloque de índices) puede tener un puntero a otro bloque de índices con más punteros que apuntan a otros bloques del archivo o puede tener valor nulo si no hay más bloques.
- *Esquema multinivel*: se tiene un bloque de índices que apunta a los bloques de índices, que a su vez apuntan a los bloques de archivos. Debe pasarse por dos niveles de bloques de índices para acceder a un bloque de archivo.
- *Esquema combinado*: se tiene en el directorio del dispositivo algunos punteros del bloque de índices, por ejemplo *k*. Los primeros *n* ($n<k$) apuntan a bloques directos, los restantes (k-n) hacen referencias a bloques indirectos.

9.4.4 Rendimiento

Una dificultad de medir el rendimiento es determinar cómo se usarán. Para cualquier tipo de acceso la asignación contigua sólo requiere un acceso para obtener un bloque. Como podemos almacenar fácilmente la dirección inicial de un archivo, podemos calcular de inmediato la dirección en disco de cualquier bloque (o el siguiente) y leerlo directamente.

Con la asignación enlazada, es posible conservar en memoria la dirección del bloque siguiente y leerlo en forma directa. Pero para llegar a un bloque *i*, con el acceso directo, pueden ser necesarias *i* lecturas del disco.

Algunos sistemas permiten acceso directo con asignación contigua y acceso secuencial con la enlazada, por eso al crear un archivo hay que declarar el tipo de acceso que se usará.

La asignación indexada es más compleja. Si el bloque de índices ya se encuentra en memoria, el acceso puede efectuarse directamente. Sin embargo, se requiere mucho espacio para almacenar en memoria el bloque de índices, y se este espacio no está disponible, primero tendremos que leer el bloque de índices y luego el bloque de datos deseado.

El rendimiento de la asignación indexada depende de la estructura de índices, tamaño de archivo y de la posición del bloque.

Algunos sistemas combinan la asignación indexada con la contigua, al utilizar esta última para archivos pequeños. Su rendimiento promedio es bueno si la mayoría de los archivos son pequeños.

9.5 Planificación del Disco

9.5.1 Planificación FCFS

Esta planificación de servicio es por orden de llegada. Generalmente no ofrece el mejor tiempo promedio de servicio.

9.5.2 Planificación SSTF

Sirve juntas todas las solicitudes próximas a la posición actual del disco, antes de desplazar la cabeza a un punto lejano para atender otra solicitud; primero la de menor tiempo de posicionamiento. Esta selección se hace a partir de la posición actual de la cabeza.

Implementamos estas estrategias moviendo la cabeza a la pista más próxima de la cola de solicitudes.

El inconveniente es que se puede tener un bloqueo indefinido para algunas solicitudes.

9.5.3 Planificación SCAN

La cabeza barre el disco de un extremo al otro atendiendo solicitudes al llegar a cada pista. Luego invierte la dirección del movimiento de la cabeza y continúa el servicio.

Suponiendo una distribución uniforme de las solicitudes de pistas, considere la densidad de solicitudes cuando la cabeza llega a un extremo e invierte la dirección. En ese momento hay pocas solicitudes para la zona inmediata detrás de la cabeza, ya que estas pistas acaban de servirse. La mayor densidad de solicitudes se encuentran en el extremo opuesto del disco. Así mismo, estas solicitudes son las que más han esperado.

9.5.4 Planificación C-SCAN

Una variante del SCAN, diseñada para ofrecer un tiempo de espera más uniforme.

La cabeza se mueve de un extremo del disco al otro, sirviendo solicitudes en su marcha, y al llegar al fin del disco regresa de inmediato al inicio del mismo sin servir ninguna solicitud en el camino.

9.5.5 Planificación LOOK y C-LOOK

SCAN y C-SCAN nunca se implementan en la práctica. Es más común que la cabeza se mueva hasta la última solicitud en cada dirección. Si ya no existen solicitudes en la dirección actual, se invierte el movimiento.

9.6 Mejoras en el rendimiento y la confiabilidad

Los discos son un cuello de botella en el rendimiento y confiabilidad de un sistema.

Una mejora técnica en la utilización de discos es la idea de varios discos que trabajan cooperativamente. Para mejorar la velocidad, los discos se separan (o intercalan). Un grupo de discos se maneja como una unidad de almacenamiento, separando los bloques en varios sub bloques, cada uno de los cuales se almacena en un disco distinto. El tiempo necesario para transferir un bloque a memoria se reduce constantemente, ya que los discos transfieren sus sub bloques en paralelo. Esto se mejora aún más si los discos se sincronizan para que no haya tiempo de latencia entre el acceso a sub bloques de cada unidad. Si además se emplean varios discos pequeños y económicos en vez de unos pocos discos grandes y costosos, la tasa de acceso mejora al aumentar el número de accesos directos por segundo.

Esta idea fue la base para desarrollar lo que se denomina grupos redundantes de discos de bajo costo (RAID), que mejoran el rendimiento, sobre todo la relación costo-rendimiento y ofrecen un medio para duplicar los datos y mejorar la confiabilidad. La velocidad mejora debido al uso de varios discos y controladores.

9.7 Jerarquía de Almacenamiento

Los discos son sólo uno de varios sistemas de almacenamiento, ya que también existen registros, memoria principal, tambores, cintas, etc. Cada sistema de almacenamiento ofrece las funciones básicas para almacenar datos y mantenerlos hasta que se recuperen más tarde; las diferencias principales entre ellos son la velocidad, costo, tamaño y volatilidad.

Todos estos medios pueden clasificarse jerárquicamente de acuerdo a su velocidad y costo.

El diseño de un sistema completo de memoria trata de equilibrar estos factores: usa sólo la cantidad imprescindible de memoria costosa, a la vez que ofrece toda la memoria barata.

10. Sistemas de archivos

10.1 Organización del sistema de archivos

Para que un sistema de computación sea fácil de usar, el **SO** ofrece una perspectiva lógica uniforme del almacenamiento de la información. EL **SO** abstrae las propiedades físicas de sus dispositivos de almacenamiento para definir una unidad lógica de almacenamiento: el archivo. El **SO** establece una correspondencia entre los archivos y los dispositivos físicos.

El sistema de archivos está formado por dos partes: el conjunto de los archivos y la estructura de directorios.

10.1.1 Concepto de archivo

Un archivo es un conjunto de información relacionada definido por su creador. La referencia al archivo, el usuario la realiza por medio de su nombre. Tiene otras propiedades: tipo de archivo, hora y fecha de creación y/o modificación, nombre del creador, longitud, etc.

Un factor que hay que tener en cuenta es si el **SO** debe conocer y apoyar los tipos de archivos. Si los soporta debe contener código para apoyar todos los tipos de archivo correctamente.

El otro extremo consiste en no imponer ni apoyar ningún tipo de archivo en el **SO**, teniendo mayor flexibilidad, pero cada programa de aplicación debe incluir su propio código para interpretar un archivo de entrada según la estructura correcta.

Los archivos generalmente son almacenados en disco. Es poco probable que el tamaño del registro físico coincida con el del registro lógico. La solución común a este problema es el empaquetamiento de varios registros lógicos en bloques físicos.

El tamaño del registro lógico, el del registro físico y la técnica de empaquetamiento determina cuántos registros lógicos hay en cada bloque físico.

Al asignar el espacio en disco en bloques, generalmente se puede desperdiciar una porción del último bloque de cada archivo. Los bytes desperdiciados, representan la fragmentación interna. Cuanto mayor el tamaño del bloque, mayor fragmentación interna.

10.1.2 Estructura de directorios

En un sistema de computación los archivos se representan mediante entradas en un directorio de dispositivo o tabla de contenido del volumen. Esto puede ser suficiente en un sistema monousuario, pero al aumentar la cantidad de almacenamiento y el número de usuarios, se hace cada vez más difícil que éstos organicen y controlen todos los archivos. La solución a este problema consiste en imponer una estructura de directorios en el sistema de archivos. Esta estructura ofrece un mecanismo para organizar los archivos en un sistema y puede sobrepasar los límites de los dispositivos e incluir distintas unidades de discos de computación diferentes.

Muchos sistemas tienen en realidad dos estructuras distintas: el directorio del dispositivo y los directorios de archivos. El primero se almacena en cada dispositivo físico y describe todos los archivos que se encuentran en él, principalmente las propiedades físicas. Los directorios de archivos son una organización lógica de los archivos en todos los dispositivos y sus entradas tienen que ver con las propiedades lógicas.

Información que se conserva para cada archivo en el directorio:

- **Nombre del archivo**: el nombre simbólico es la única información que se mantiene en forma legible para el usuario.
- **Tipo de archivo**: información necesaria para sistemas que soportan distintos tipos.
- **Ubicación**: Puntero al dispositivo y a la ubicación del archivo en ese dispositivo.
- **Tamaño**: tamaño actual del archivo y, posiblemente, el tamaño máximo permitido.
- **Posición actual**: puntero a la posición actual de lectura / escritura en el archivo.
- **Protección**: información sobre el control de acceso.

- **Recuento de uso**: indica el número de procesos que actualmente lo están usando.
- **Hora, fecha e identificación de proceso**: esta información puede conservarse para la creación, la última modificación y para el último acceso. Estos datos pueden servir para la protección y la supervisión de la utilización.

En un sistema con gran cantidad de archivos, el tamaño del directorio puede ser de varios miles de bytes, por lo que es probable que el mismo tenga que almacenarse en su dispositivo y traerse a memoria por partes, según se necesite.

El directorio del sistema se puede considerar como una tabla de símbolos que traduce los nombres de archivo a sus entradas en el directorio.

Una lista de entradas del directorio requiere una búsqueda secuencial para encontrar una entrada determinada. Para reutilizar la entrada del directorio, existen varias alternativas: podemos marcar la entrada como "no utilizada", o podemos añadirla a una lista de entradas libres para el directorio. Otra alternativa consiste en copiar la última entrada en la localidad liberada y reducir el tamaño del directorio. También puede emplearse una lista ligada para disminuir el tiempo necesario para eliminar un archivo. La gran desventaja de una lista lineal de entradas de directorio es la búsqueda secuencial necesaria para encontrar un archivo.

Una lista ordenada permite una búsqueda binaria que reduce el tiempo promedio de búsqueda. Siempre hay que mantener una lista ordenada. Este requisito puede complicar la creación de archivos y eliminación de archivos. En este caso puede servir de ayuda un árbol binario enlazado.

Otra estructura utilizada es la tabla de dispersión, que puede mejorar considerablemente el tiempo de búsqueda. La inserción y eliminación son bastantes directas, aunque hay que tomar medidas para evitar colisiones, situaciones donde dos nombres de archivos se traducen a la misma localidad. Las principales dificultades con una estructura de este tipo son el tamaño generalmente fijo de la tabla y la dependencia de la función de dispersión respecto al tamaño de la tabla.

10.2 Operaciones sobre archivos

El **SO** nos ofrece llamadas al sistema para crear, escribir, leer, reestablecer y eliminar archivos.

Qué es lo que hace el **SO** en:

Creación de archivos: se requieren dos pasos. Primero hay que encontrar espacio libre; segundo hay que anotar el nuevo archivo en el directorio.

Escritura de archivos: se efectúa una llamada al sistema especificando el nombre del archivo y la información que se escribirá en él. Al recibir el nombre del archivo, el sistema busca en el directorio para determinar su ubicación. La entrada del directorio tendrá que almacenar un puntero al bloque actual del archivo (generalmente el inicial) con este puntero podemos calcular la dirección del siguiente bloque y escribir la información. Hay que actualizar el puntero de escritura y, así, se puede escribir una serie de bloques del archivo mediante escrituras sucesivas.

Lectura de archivos: usamos una llamada al sistema especificando el nombre del archivo y el lugar en memoria donde debe colocarse el siguiente bloque. Una vez más, se busca en el directorio la entrada correspondiente y se necesitará un puntero al siguiente bloque que se leerá. Una vez más que se ha leído el bloque, se actualiza el puntero.

Reposicionamiento de archivos: se busca en el directorio la entrada indicada y la posición actual se modifica de manera que apunte al inicio del archivo. Es posible que no se lleve a cabo E/S para el reposicionamiento.

Eliminación de archivos: para eliminar un archivo buscamos su nombre en el directorio. Una vez localizado, liberamos todo el espacio del archivo e invalidamos la entrada del directorio.

Todas las operaciones implican la búsqueda en el directorio de la entrada al archivo especificado. Para evitar búsquedas constantes, muchos sistemas abren el archivo cuando se usa por primera vez y el **SO** mantiene una pequeña tabla con información de todos los

archivos abiertos. Cuando el archivo deja de usarse, se cierra y se elimina de la tabla de archivos abiertos.

10.3 Métodos de Acceso

10.3.1 Acceso secuencial

La información del archivo se proceso registro tras registro.

Una operación de lectura/escritura lee/escribe la siguiente porción del archivo y el puntero avanza automáticamente.

10.3.2 Acceso directo

Este método se basa en el disco como modelo de archivo. Se considera que el archivo es una secuencia numerada de bloques o registros. Un archivo de acceso directo permite leer o escribir cualquier bloque, sin restricciones en cuanto al orden de lectura o escritura, es muy útil para obtener grandes cantidades de información.

10.3.3 Otros métodos de acceso

A partir del acceso directo es posible construir otros métodos de acceso. Estos, generalmente implican la construcción de un índice para el archivo. Para encontrar una entrada en el archivo, buscamos primero en el índice y luego usamos el puntero para acceder directamente al archivo y encontrar la entrada deseada.

Si el archivo no es de gran tamaño, podemos conservar el índice en memoria, y efectuar una búsqueda (binaria) en el mismo, lo que nos permite encontrar el bloque que contiene la entrada deseada para iniciar el acceso. Realizamos así, una búsqueda con pocas E/S.

El inconveniente se genera en archivos de gran tamaño. Una solución es crear un índice primario, que contenga los punteros a un índice secundario, que contenga punteros a los elementos de datos.

10.4 Semántica de consistencia

La semántica de consistencia es un criterio importante para la evaluación de cualquier sistema que permita compartir archivos. Se trata de una característica del sistema que especifica la semántica de especificar qué modificaciones realizadas por un usuario en los datos pueden ser observados por otros usuarios.

10.4.1 Semántica de UNIX

Las escrituras que efectúa un usuario abierto son visibles de inmediato para los otros usuarios que tengan el archivo abierto al mismo tiempo.

Existe un método para compartir archivos donde los usuarios comparten el puntero a la posición actual en el archivo, de modo que, si un usuario avanza el puntero, afecta a todos los usuarios que comparten el archivo. En este caso, el archivo tiene una imagen única que intercala todos los accesos, sin importar si *origen*.

10.4.2 Semántica de sesiones

Las escrituras que efectúa un usuario en un archivo abierto no son inmediatamente visibles para los demás usuarios que tienen el archivo abierto al mismo tiempo.

Una vez que se ha cerrado un archivo, los cambios efectuados sólo son visibles para las sesiones que comiencen más tarde. Las instancias del archivo que ya están abiertas no reflejan estos cambios.

Así, los usuarios pueden llevar a cabo accesos de Lectura y Escrituras concurrentemente sobre su imagen del archivo, sin demoras. Observe que se aplican muy pocas restricciones a la planificación de los sucesos.

10.4.3 Semántica de archivos compartidos inmutables

Una vez que el creador declara un archivo como compartido, ya no puede modificarlo. Un archivo inmutable tiene dos propiedades importantes: su nombre no se

puede volver a utilizar y no es posible alterar su contenido. Es muy sencillo implementar esta semántica en un sistema distribuido, ya que el compartimiento es muy disciplinado.

10.5 Organización de estructuras de directorio

Operaciones que se aplican al directorio:
Búsqueda.
Creación de archivos.
Eliminación de archivos.
Listado de directorio
Copia de seguridad.

10.5.1 Directorio de un solo nivel

Todos los archivos se conservan en el mismo directorio.

Tiene limitaciones cuando aumenta el número de archivos (debiendo tener nombres únicos) o hay más de un usuario. Cada vez se hace más difícil recordar todos los nombres para crear uno nuevo.

10.5.2 Directorio de dos niveles

Cada usuario tiene su propio directorio de archivos (DAU), cada uno con una estructura similar. Cuando se inicia un trabajo de usuario o se conecta al sistema, se busca en el directorio de archivos maestro (MFD) del sistema que está indexado por nombre de usuario o número de cuenta, y cada entrada apunta al directorio correspondiente al usuario.

Toda operación a un archivo, implicará una referencia al DAU el usuario correspondiente. Para crear o eliminar los directorios del usuario se ejecuta un programa especial (obviamente restringido para los usuarios).

Un problema que presenta es el aislamiento del usuario. Algunos sistemas no permiten que se tenga acceso a los archivos en el directorio de otro.

Si se permite el acceso, el usuario debe tener la capacidad de nombrar un archivo en otro directorio.

Para los programas del sistema, se define un directorio de usuario especial que contiene los archivos de sistema. Cuando se proporciona un nombre de archivo para cargarlo, el **SO** busca primero en la DAU del usuario, si no lo encuentra, automáticamente busca en el directorio especial que contiene los archivos de sistema. La secuencia de directorios que se buscan al nombrar un archivo se denomina *ruta de búsqueda*.

10.5.3 Directorio con estructura de árbol

Un directorio (o subdirectorio) contiene un conjunto de archivos y/o subdirectorios. Todos los directorios tienen el mismo formato interno. Un bit en cada entrada del directorio define tal entrada como un archivo (0) o un subdirectorio (1). Se emplean llamadas especiales al sistema para crear o eliminar directorios.

Durante el uso normal, cada usuario tiene un *directorio actual*. Cuando se efectúa una referencia a un archivo se busca en el directorio actual, si no se encuentra allí, el usuario debe especificar una ruta o cambiar su directorio actual.

Los nombres de ruta pueden ser de dos tipos: absolutos o relativos.

Para eliminar un directorio que no está vacío, tenemos dos alternativas:

1. El sistema no permite borrarlo, entonces hay que borrar todos los archivos y subdirectorios dentro, vaciándolos antes.
2. El sistema borrar el directorio junto con los archivos y subdirectorios.

10.5.4 Directorios de grafo acíclico

Consideremos el caso de dos programadores que trabajan conjuntamente en un proyecto. Los archivos relacionados con este proyecto podrían estar en un subdirectorio, separados de otros proyectos y de los archivos de cada programador. Ambos programadores desearán que el subdirectorio se encuentre en sus directorios. Una salida a este problema es que le subdirectorio puede compartirse.

Un grafo *acíclico* permite que los directorios contengan subdirectorios y archivos compartidos.

Se pueden implementar de varias maneras. Una de las más comunes consiste en crear una nueva entrada llamada *enlace*. Un enlace es un puntero a otro archivo o subdirectorio, se puede implementar como una **ruta absoluta o relativa**.

Otra implementación consiste en duplicar la información relacionada con esos archivos en los directorios que los comparten, con los que se obtienen entradas idénticas.

Una estructura de directorio de grafo acíclico es más flexible que una estructura de árbol.

Si *eliminamos* un archivo compartido. Pueden quedar punteros sueltos de un archivo inexistente o, peor aún, si ese espacio fue utilizado para grabar otro archivo los punteros sueltos pueden apuntar a algún lugar de otro archivo.

Ante estas situaciones, si la implementación fue con enlaces, al eliminar un enlace no se afecta el archivo original. Si se borra la entrada correspondiente al archivo, se libera su espacio del archivo y los enlaces quedan sueltos. Es posible buscar todos los enlaces y eliminarlos, pero debemos tener una lista de los enlaces asociados. Otra alternativa es dejar los enlaces hasta que alguien intente usarlos, si no existe el nombre del archivo al cual apunta el enlace, se lo trata como una referencia de nombre de archivo ilegal.

Otra estrategia para la eliminación consiste en conservar el nombre del archivo hasta que se eliminen todas sus referencias. Para implementar este procedimiento debemos contar con algún mecanismo que determine si la referencia eliminada era la última para ese archivo. Para esto nos es suficiente con un recuento del número de referencias.

10.5.5 Directorio de grafo general

Un serio problema que se presenta al emplear una estructura de grafo acíclico consiste en asegurar que no existan ciclos.

Si se permite que haya ciclos en el directorio, también queremos evitar la doble búsqueda de un mismo componente, tanto por razones de corrección como de rendimiento. Un algoritmo mal diseñado puede provocar un ciclo infinito.

Un recuento de referencia 0 indica que ya no hay más referencias al archivo o directorio. Aunque es posible, cuando existe un ciclo, que el recuento de referencias sea distinto de 0. Esta anomalía es el resultado de una autorreferencia (un ciclo) en la estructura del directorio. En estos casos normalmente es necesario utilizar la *recolección de basura* para determinar cuándo se borrar la última referencia y así poder reutilizar el espacio libre. La recolección de basura implica recorrer el sistema de archivos y marcar todo aquello a lo que se pueden acceder; luego, en un segundo paso recopila todo lo que no está marcado para incluirlo en una lista de espacio libre.

Lo difícil es evitar ciclos al añadir nuevos enlaces a la estructura. Para esto, se cuenta con algoritmos para detectar ciclos en grafos, pero los cálculos son muy costosos, sobre todo cuando el grafo se encuentra en disco.

10.6 Protección de archivos

A la información almacenada en un sistema de computación debemos protegerla de daños físicos (confiabilidad) y del acceso inadecuado (protección).

La confiabilidad se obtiene creando copias de los archivos.

La necesidad de protección surge de la capacidad de acceder a los archivos.

Los mecanismos de protección ofrecen acceso controlado limitando los tipos de acceso a archivos que pueden efectuarse. El acceso se permite o niega dependiendo de varios factores, uno de los cuales es el tipo de acceso solicitado. Es posible controlar varas clases de operaciones:

Lectura.
Escritura.
Ejecución.
Adición.
Eliminación.

Las operaciones con directorios que deben proteger son algo diferentes. Puede que el conocimiento de la existencia y nombre de un archivo sea significativo por sí mismo, por ello, la visualización del contenido de un directorio debe ser una operación protegida.

La protección puede estar asociada al archivo o a la ruta que se usa para especificarla.

10.6.1 Nominación

Los esquemas de protección de varios sistemas dependen de que el usuario no tenga acceso a un archivo que no puede nombrar. Este esquema supone que no hay ningún mecanismo para obtener los nombres de los archivos de otros usuarios y que es difícil adivinarlos.

10.6.2 Contraseñas

Otra estrategia consiste en asociar una contraseña con cada archivo.

Desventajas:

Si asociamos una contraseña con cada archivo, puede ser muy grande el número de contraseñas para recordar. Si se emplea la misma contraseña para todos los archivos, una vez que ésta se descubre, todos los archivos se hacen accesibles. Algunos sistemas, permiten que el usuario asocie la contraseña con un subdirectorio.

10.6.3 Lista de acceso

Otra estrategia es hacer que el acceso dependa de la identidad del usuario. Una *lista de acceso* que especifique los nombres de los usuarios y los tipos de acceso permitidos para cada uno.

El inconveniente con las listas es su longitud. Tiene consecuencias indeseables:

La construcción de la lista puede ser tediosa e infructuosa, especialmente si no conocemos por adelantado la lista de usuarios del sistema.

La entrada del directorio, que antes era de tamaño fijo, ahora necesita ser de tamaño variable, lo que complica la administración del espacio.

10.6.4 Grupos de acceso

Los problemas anteriormente planteados pueden resolverse usando una versión condensada de la lista de acceso. Para realizar esto, muchos sistemas reconocen tres clasificaciones de los usuarios en relación con cada archivo:

- Dueño: el usuario que creó el archivo.
- Grupo (o grupo de trabajo): es un conjunto de usuarios que comparten el archivo y necesitan acceso similar.
- Otros.

Para que este esquema funcione correctamente, hay que controlar con cuidado la pertenencia a los grupos.

Con frecuencia, cada campo es un conjunto de bits, cada uno de los cuales permiten o impide el acceso que con él se relaciona.

10.7 Aspectos de la implementación

Debemos considerar dos cuestiones de diseño en un sistema de archivos. Una es el aspecto que presentará este sistema al usuario. La otra es la creación de algoritmos y las estructuras de datos para relacionar el sistema de archivos lógico con los dispositivos destinados a usarse con ese sistema de archivos.

Niveles del sistema:

Programas de aplicación
Sistema lógico de archivos
Módulo de organización de archivos
Sistema básico de archivos
Control de E/S
Dispositivos

El nivel inferior, *control de E/S*, consiste manipuladores de dispositivos y de interrupciones para transferir información entre la memoria y el sistema de disco. El *sistema básico de archivos* emplea esta información para leer y escribir bloques concretos al disco.

El *módulo de organización de archivos* tiene conocimientos de los archivos y los bloques en disco.

El módulo de organización de archivos puede generar las direcciones de los bloques que tiene que leer el sistema básico de archivos. Por último, el *sistema lógico de archivos* usa la estructura del directorio para proporcionar al módulo de organización de archivos los valores que necesita, a partir de un nombre simbólico de archivo.

Otro aspecto de la utilización de archivos que se debe definir es el control de acceso. Hay algunos diseñados para comprobar la protección del archivo únicamente al abrirlo.

Aunque el **SO** puede comprobar la protección en este punto, debe hacerlo continuamente para que no se pueda escribir en un archivo abierto para lectura; por lo tanto, hay que comprobar la validez de *cada acceso*.