Sistemas Operativos

Teórico

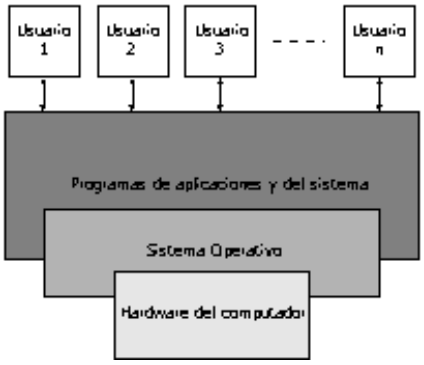
Autor Original: **Mónica Canto**

1. Introducción

Un sistema operativo es un programa que actúa como intermediario entre el usuario y la máquina. El primer objetivo es hacer el sistema cómodo de usar, tratado de hacerlo de la forma más eficiente posible.

1.1 ¿Qué es un sistema operativo?

Los sistemas se dividen en cuatro componentes: El hardware, el sistema operativo, los programas de aplicación y los usuarios.



El sistema operativo controla y coordina el uso del hardware entre los diversos programas de aplicación. El sistema de computación tiene muchos recursos, y el sistema operativo actúa como gestor de esos recursos para los distintos usuarios, decidiendo que solicitudes atenderá inmediatamente, para operar de manera eficiente y justa.
Controla la ejecución de los programas de usuario a fin de evitar errores y el uso incorrecto del computador.
Hay distintos programas de usuario, pero todos tienen operaciones en común, que son las encargadas de la E/S. Estas operaciones se agrupan en un solo programa, que es el sistema operativo.

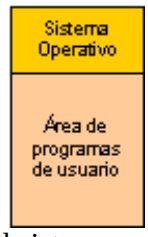
Otra definición de sistema operativo es el programa individual (núcleo o kernel) que se está ejecutando todo el tiempo en el computador.

Los objetivos del sistema operativo son:

- *Comodidad:* Facilitan las tareas de cómputo.
- *Eficiencia:* Se desea hacer un sistema que aproveche al máximo los recursos.

Los SO se crearon para facilitar el uso del hardware y estudiando los problemas se dio pie a la introducción de nuevas características en el hardware, nuevas arquitecturas.

1.2 Sistemas por lotes sencillos

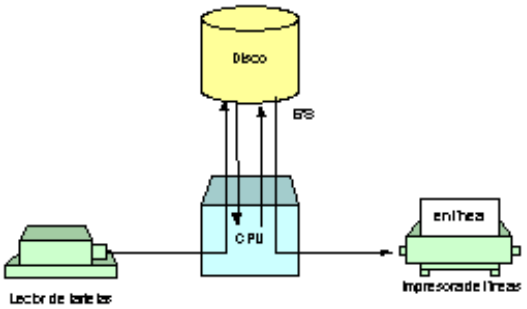


Los primeros computadores eran máquinas enormes que se controlaban desde una consola. Los dispositivos de entrada eran lectores de tarjetas, y los de salida, impresoras de línea o perforadores de tarjetas.
Los usuarios no interactuaban directamente con los computadores, sino que preparaban los trabajos en tarjetas perforadas, lo entregaban al operador del computador y en algún momento posterior, obtenía los resultados, o un vuelco de memoria (contenido de los registros) si había errores.

El sistema operativo era sencillo, y estaba siempre residente en memoria. Lee tarjetas, procesa e imprime, pero hay falta de interacción con el usuario.

En este entorno de ejecución, la CPU con frecuencia está ociosa, debido a que los dispositivos E/S mecánicos son lentos. Los dispositivos de lectura, fueron mejorando con los años, pero también la CPU, por lo tanto el problema aumentó aún más.

La introducción de la tecnología de discos magnéticos fue útil en este sentido. En vez de leer las tarjetas al CPU, se leen al disco. Luego el CPU lee el disco para obtener las entadas y guarda en el mismo las salidas, las cuales son leídas luego por la impresora. Esto es llamado “Spooling” *(simultaneous peripheral operation on-line)*.
El disco se usa como buffer, para leer por adelantado hasta donde sea posible de los dispositivos de entrada, y para guardar los archivos de salida hasta que los dispositivos de salida puedan aceptarlos, mientras la CPU se mantiene procesando continuamente, sin esperar entradas o salidas. También sirve para procesar datos en sitios remotos.



1.3 Sistemas por lotes multiprogramados

En el caso anterior, no se puede mantener ni los dispositivos E/S ni el CPU ocupados todo el tiempo, ya que si los trabajos llegan directo de la lectora, se deben ejecutar en orden de llegada, en cambio si están en un disco, el SO puede utilizar *multiprogramación* la cual aumenta el aprovechamiento de la CPU, organizando los trabajos para que la CPU tenga siempre uno para ejecutar.
La idea es que el SO mantiene varios trabajos en la memoria a la vez, manteniendo el resto en reserva. Luego escoge uno de los que están en memoria y lo ejecuta. Mientras ese trabajo tiene que esperar, por ejemplo terminar una operación E/S, en un sistema sin multiprogramación, la CPU estaría ociosa. En uno multiprogramado, el sistema selecciona otro trabajo y lo ejecuta. En algún momento el primer trabajo termina su espera y recupera la CPU.
Los SO multiprogramados son relativamente complejos, ya que deben tomar decisiones.

1.4 Sistemas de tiempo compartido

Los sistemas por lotes tienen problemas desde el punto de vista del usuario, puesto que éste no puede interactuar con el trabajo durante su ejecución, y debe preparar las tarjetas para todos los resultados posibles, ya que en un trabajo de varios pasos, algunos pasos pueden depender del éxito de los anteriores. Ej.: La ejecución depende de una compilación exitosa.

En un sistema *multitarea* o *interactivo*, se ejecutan múltiples trabajos mientras la CPU conmuta entre ellos, pero la conmutación es tan frecuente que los usuarios pueden interactuar con cada programa durante su ejecución. El usuario da instrucciones al SO o programa y recibe una respuesta inmediata. Por lo regular se usa el teclado para entrada, y la pantalla para la salida.

Para que los usuarios accedan con comodidad a los datos, deben contar con un sistema de archivos. Estos se organizan en directorios, facilitando su localización y acceso. El SO controla el acceso a archivos por parte de los usuarios.

En los sistemas de *tiempo compartido* cada usuario tiene por lo menos un programa individual en la memoria (conocido como *proceso*). La E/S interactiva se efectúa a un ritmo humano, limitadas por la velocidad de respuesta del usuario, la cual es lenta para la CPU. Mientras la CPU esta ociosa por esta causa, el SO conmuta la CPU a otro usuario.
Los usuarios comparten la CPU simultáneamente, y como el computador cambia con rapidez de un usuario al siguiente, cada uno recibe la impresión de que tiene su propio computador.

Al igual que en la multiprogramación, se necesita mantener varios trabajos simultáneamente en la memoria, por lo que requiere gestión y protección de memoria. El disco funciona como *memoria virtual*, permitiendo ejecutar trabajos que no caben en la memoria principal.

1.5 Sistemas de computador personal

Con la caída de costos de hardware, se volvió factible tener un sistema dedicado a un solo usuario. Los dispositivos E/S cambiaron, apareció el Mouse, nuevas impresoras, etc.

Al comienzo, los SO para PC no eran ni multiusuario ni multitarea, y no tenían protección de memoria. Los SO más modernos, en lugar de tratar de aprovechar al máximo la CPU, optan por maximizar la comodidad del usuario, y la rapidez con la que responden a sus necesidades.

Aunque aparentemente, la protección de archivos no es necesaria en las PC, es común que los computadores se vinculen con otros a través de Internet, por lo que la protección de archivos vuelve a ser necesaria, por ello el DOS era tan propenso a los programas mal intencionados (gusanos, virus, etc.).

Con el paso del tiempo, las características de los sistemas operativos para macrocomputadores, se fueron ajustando a los computadores personales.

1.6 Sistemas paralelos

Tienen más de un procesador, los cuales comparten el bus, reloj, memoria y periféricos (están *fuertemente acoplados*). Cuando usamos n procesadores, esperamos un aumento de n en *rendimiento* o velocidad, pero no es así, es menor que n, porque deben compartir los recursos.

Los multiprocesadores pueden ahorrar dinero, ya que si varios usuarios usan los mismos archivos o programas, es más económico tener un único disco con ellos, que varios discos, uno para cada usuario.
También mejoran la confiabilidad, ya que si uno falla, no se detiene el sistema, sino que los demás se encargan de procesar. Es un sistema *tolerante a fallos*. Pero para que esto funcione, debe haber un mecanismo de detección y corrección de fallos.

Los sistemas multiprocesador actuales siguen el modelo de *multiprocesamiento simétrico*, ya que cada procesador ejecuta una copia idéntica del SO, y se comunican entre si. Los sistemas de *multiprocesamiento asimétrico*, asignan a cada procesador una tarea específica. Hay maestros y esclavos, pero es necesario controlar cuidadosamente la E/S, para asegurar que los datos lleguen al procesador apropiado.

A medida que baja el precio y aumenta la potencia de los microprocesadores, se comienzan a utilizar esclavos, como por ejemplo un administrador de disco, o de E/S, que le ahorran trabajo al CPU principal.

1.7 Sistemas distribuidos

Distribuyen el cómputo entre varios procesadores, que no comparten ni la memoria ni el reloj. Cada procesador tiene su memoria local y se comunican entre si por buses. Son sistemas que están *débilmente acoplados*. Los procesadores pueden tener diferentes tamaños y funciones.

Las ventajas de los sistemas distribuidos son:

- **Recursos compartidos:** Un usuario de un sitio, puede aprovechar los recursos disponibles de otro. Hay mecanismos para compartir archivos en sitios remotos.
- **Computación más rápida:** Divide los cálculos en subcálculos y los distribuye entre los sitios para procesar concurrentemente (llamado *carga compartida*).
- **Confiabilidad:** Si un sitio falla, los demás pueden seguir funcionando.
- **Comunicación:** Los usuarios pueden transferir archivos o comunicarse entre si por correo electrónico, y los programas pueden intercambiar datos con otros programas del mismo sistema.

1.8 Sistemas de tiempo real

Se utiliza cuando los requisitos de tiempo para una operación son estrictos, tiene restricciones de tiempo bien definidas.

Hay dos tipos:

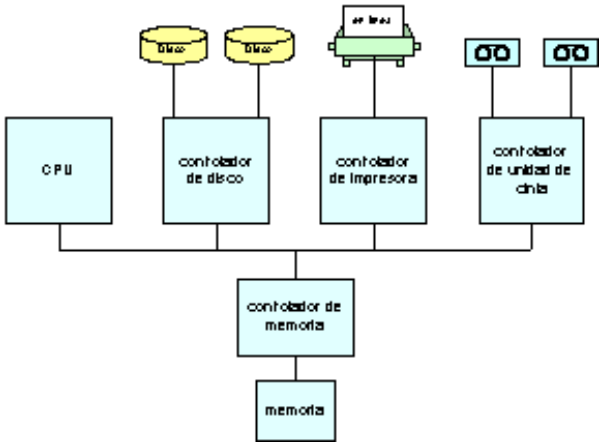
- **Tiempo real duro:** Las tareas críticas se deben terminar a tiempo. Todos los retardos del sistema están limitados, por lo tanto los datos se deben guardar en memoria rápida como ROM, además carecen de la mayor parte de las funciones avanzadas de los sistemas operativos, como por ejemplo, memoria virtual.
- **Tiempo real blando:** Una tarea crítica tiene prioridad respecto a otras, y se deben evitar los retardos, pero no apoyan el cumplimiento estricto de plazos.

2 Estructuras de los sistemas de computación

El SO debe asegurar el correcto funcionamiento del sistema de computación. El hardware debe contar con los mecanismos para que los programas de usuario no interfieran con la operación correcta del sistema.

2.1 Funcionamiento de los sistemas de computación

Un sistema de computación consta de una CPU, y varios controladores de dispositivos (cada uno encargado de un tipo de dispositivo específico), conectados por un bus común, el cual ofrece acceso a la memoria compartida. La CPU y los controladores de dispositivos funcionan de manera concurrente, compitiendo por los ciclos de memoria, utilizando un controlador de memoria, para sincronizar el acceso.
Para que un computador comience a funcionar necesita tener un programa inicial que ejecutar. Este *programa de arranque*, suele ser sencillo; asigna valores iniciales a todos los aspectos del sistema, desde los registros de la CPU, hasta los controladores de dispositivos y la memoria. Luego carga en memoria el Kernel, a fin de comenzar la ejecución del SO.
El hardware, puede generar una *interrupción* en cualquier momento, enviando una señal a la CPU, por lo regular, a través del bus. El software solo puede hacerlo ejecutando una función especial denominada *llamada al sistema*.



Se cuenta con una rutina de servicio para cada una de las interrupciones, la cual se encarga de atenderla. Cuando la CPU se interrumpe, suspende lo que estaba haciendo y trasfiere la ejecución a una posición fija de memoria, que contiene la dirección inicial de la rutina de esa interrupción. La rutina se ejecuta, y al terminar, la CPU reanuda lo que estaba haciendo.
En general, la tabla con direcciones a rutinas, se almacena en la memoria baja (primeras 100 posiciones). Es llamado el “vector de interrupciones”, y la rutina es identificada por un número de dispositivo único, incluido en la solicitud de la interrupción.

La arquitectura de interrupciones debe guardar la dirección de la instrucción interrumpida. Antes se guardaba en un lugar fijo, permitiendo una única interrupción a la vez, ya que si se volvía a interrumpir, se sobrescribía el valor anterior, y se perdía la instrucción donde había quedado el programa antes de la primera interrupción. Actualmente se guardan en la pila del sistema.

Por lo regular, cuando se atiende una interrupción, se desactivan las demás hasta que la actual termina. Luego se vuelven a activar. Las arquitecturas más avanzadas, permiten ejecutar varias a la vez, empleando un esquema de prioridades.

Los SO modernos son *controlados por interrupciones*. Ya que el sistema espera, hasta que suceda algo, como la invocación a un procedimiento, o dispositivos de E/S que atender, etc.

Una *trampa* (excepción) es una interrupción generada por software, indicando un error (división entre cero, acceso a memoria inválido, etc.) o la solicitud de un servicio del SO por parte de un programa.
Cuando ocurre una excepción, el hardware transfiere el control al SO, el SO guarda el contexto, y atiende la interrupción.

2.2 Estructura de E/S:

Un controlador de dispositivos, tiene un buffer local, y un conjunto de registros de propósito especial. El controlador se encarga de transferir datos entre el dispositivo y su buffer local, y el tamaño del buffer depende del controlador y del dispositivo.

2.2.1 Interrupciones de E/S:

Para iniciar una E/S, la CPU carga el valor apropiado en los registros del controlador, el cual los examina para saber que hacer. Esto generalmente ocurre, por un pedido de E/S de un proceso del usuario. Hay dos posibles formas de hacerlo:

- **Sincrónica:** Se inicia la E/S, y cuando finaliza se devuelve el control al usuario. Se puede hacer utilizando la operación *wait* que mantiene la CPU ocupada hasta la siguiente interrupción, o en el caso de no poseerla, ejecutando un ciclo, hasta que se levante una bandera (poner en 1 un BIT especial). Esto no permite E/S concurrente.
- **Asincrónica:** Devuelve el control sin esperar que se complete la E/S, así esta continúa mientras se realizan otras operaciones. Se mantiene una *tabla de estado de dispositivos* que controla el estado del dispositivo (apagado, ocioso, ocupado) y una *cola de espera* para cada dispositivo que guarda la lista de las solicitudes que están esperando por él. Puede haber solicitudes de E/S concurrentes, aumentando la eficiencia del sistema, ya que las E/S son lentas.

2.2.2 Estructura DMA:

La CPU demora 2 microsegundos en atender una interrupción. Los dispositivos rápidos, efectúan interrupciones cada 4 microsegundos, por lo tanto no deja tiempo para otros procesos.
Para resolver este problema, se emplea el *DMA* (*Acceso directo a memoria*) con los dispositivos de alta velocidad. El controlador del dispositivo, transfiere datos del buffer, directamente a la memoria, sin pasar por el CPU, generando una interrupción por cada bloque, y no por cada byte.

2.3 Estructura de almacenamiento

Los programas deben estar en memoria principal para ejecutarse, la cual es la única área de almacenamiento grande a la cual el procesador puede acceder directamente. Es una matriz de palabras, donde cada palabra tiene su propia dirección.
La instrucción **load** (cargar) transfiere una palabra de la memoria principal a un registro de la CPU, la **store** (guardar) hace lo contrario.

Por dos razones, los datos y programas no residen permanentemente en la memoria:

- La memoria principal generalmente es pequeña para contener todos estos datos.
- Es *volátil*, o sea, pierde su contenido cuando se interrumpe la electricidad.

Casi todos los sistemas cuentan con *almacenamiento secundario*, el cual es grande, y no volátil. Generalmente es un disco magnético.

2.3.1 Memoria principal

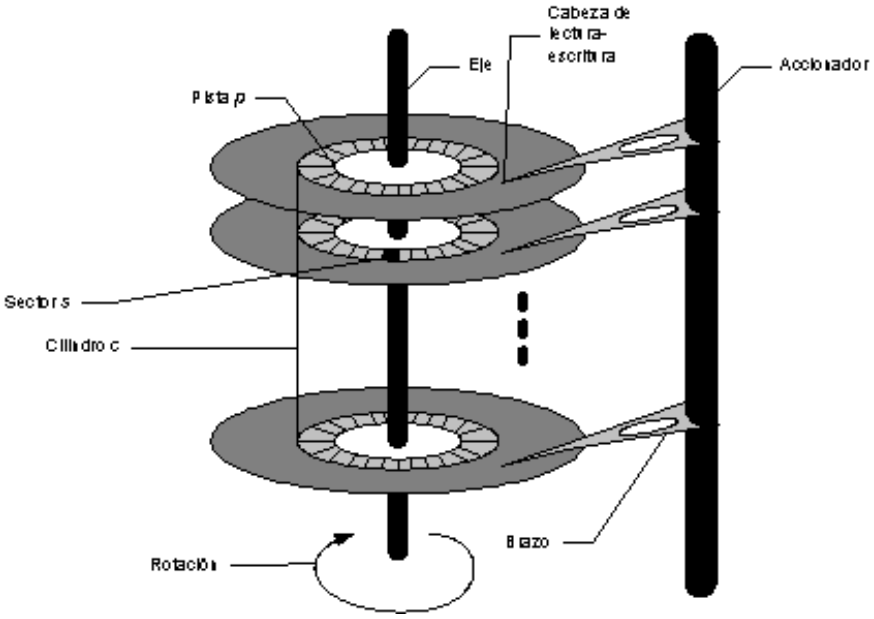
La memoria principal y los registros, son los únicos a los que la CPU accede directamente, por lo tanto, cualquier instrucción que se esté ejecutando, debe estar en ellos.

Muchas arquitecturas, a fin de facilitar la interacción con dispositivos E/S, cuentan con *E/S mapeada en memoria*, donde apartan direcciones de memoria, y se establece correspondencia entre ellas, y los registros del dispositivo, por lo tanto, escribir o leer en esa dirección, implica hacerlo también con los registros del dispositivo. Esto es útil para dispositivos de respuesta rápida como el video.
Nota: Recordar tarea de Arquitectura, donde escribíamos en memoria para trabajar con el puerto del Mouse, y desplegarlo en pantalla.

Si la instrucción está en un registro, se necesita un ciclo de reloj para ejecutarla. Si está en memoria, el CPU debe esperar mientras los datos pasan por el bus. Esto es indeseable, por lo tanto se coloca memoria rápida como intermedio llamada “caché”.

2.3.2 Discos magnéticos

Son el medio de almacenamiento secundario más común. El diámetro de los platos varía entre 1.8 y 5.25 pulgadas, y sus dos superficies están cubiertas con un material magnético.
La *cabeza de lectura escritura*, vuela sobre la superficie del plato. Las cabezas están unidas a un *brazo* del disco, que las mueve como una unidad.
La superficie del plato se divide lógicamente en *pistas* circulares que se subdividen en *sectores*. El conjunto de pistas que están en una posición del brazo, se llama *cilindro*. Cuando el disco se está usando, un motor lo hace girar a alta velocidad.
La velocidad del disco tiene dos partes:



- **Tasa de transferencia:** Es la rapidez con que los datos fluyen entre el disco y el computador.
- **Tiempo de acceso aleatorio:** Es el tiempo que demora en acceder a los datos. Se divide en:
  - **Tiempo de búsqueda:** Es el tiempo que toma mover el brazo del disco al cilindro deseado.
  - **Latencia rotacional:** Es el tiempo que el sector deseado tarda en girar hasta quedar debajo de la cabeza del disco.

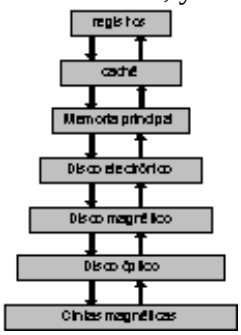
En el disco hay espacios entre cada sector. Cuando leo un sector, antes de leer el siguiente, debo calcular la paridad. Si los sectores fueran consecutivos, no tendría tiempo de calcularla y tendría que esperar una vuelta de disco antes de leer la siguiente. El controlador del disco es el que sabe donde debe leer.

Tipos de disco:

- **Removible:** Permite montar diferentes discos según sea necesario. Están protegidos por un estuche plástico.
  - Los discos flexibles son discos magnéticos removibles de bajo costo.

2.3.3 Cintas magnéticas

Fue uno de los primeros medios de almacenamiento secundario. Puede contener grandes cantidades de datos, pero el tiempo de acceso es lento, y no se puede acceder aleatoriamente, solo en forma secuencial. Sirven para el backup.



2.4 Jerarquías de almacenamiento

Los niveles más altos, son costosos pero rápidos. Además de la rapidez y el costo, está la volatilidad. Los datos deben escribirse en almacenamiento no volátil para mantenerlos seguros.

2.4.1 Uso de cachés

La información normalmente se guarda en la memoria principal. A medida que se usa, esa información se copia en un sistema de almacenamiento más rápido, como el caché, de forma temporal. Cuando necesitamos un elemento en

particular, primero vemos si está en el caché. Si es así, utilizamos la información directamente del caché, sino, utilizamos la información de la memoria, colocando una copia en el caché, suponiendo que hay una probabilidad elevada de que se vaya a necesitar nuevamente.
Los registros internos programables, funcionan de caché de alta velocidad para la memoria principal. La memoria principal se considera como un caché rápido para la memoria secundaria.
La mayor parte de los sistemas cuenta con un caché de instrucciones, sin el cual debería esperar varios ciclos mientras se trae una instrucción de la memoria principal.
Debido a el tamaño limitado del caché, su gestión es un problema de diseño importante.

2.4.2 Coherencia y consistencia

Cuando leo un archivo *A*, hago una copia en el caché, y luego en un registro interno. Si lo modifico, el valor de *A* va a variar en los distintos lugares donde se encuentra, y será igual en todos los lugares luego de que lo vuelva a guardar en el disco.
En un sistema común, esto no causa problemas, en cambio en uno multiprocesador, en el que además de mantener registros internos, la CPU cuenta también con un caché local, podría haber una copia de *A* en varios cachés simultáneamente, y como pueden trabajar concurrentemente, es preciso asegurarse de actualizar el valor de *A*, en todos los cachés. Este problema se denomina *coherencia de cachés*, y es un problema de hardware.
En un entorno distribuido, la situación se vuelve más compleja, ya que podrían guardarse varias copias del mismo archivo en diferentes computadores.

2.5 Protección por hardware

Los primeros sistemas eran monousuario, operados por el programador, el cual tenía control total sobre el sistema. A medida que se fueron desarrollando los sistemas operativos, se les comenzó a transferir el control, desempeñando principalmente las E/S.

El SO para mejorar el aprovechamiento de la CPU, comenzó a compartir los recursos del sistema entre varios programas simultáneamente. Cuando el sistema se ejecutaba sin compartir, un error en el programa podía causar problemas solo sobre él mismo. Al compartir, muchos procesos pueden verse afectados por ese error.

El hardware detecta muchos errores de programación de cuyo manejo normalmente se encarga el SO. Cuando detecta ese error, transfiere el control al SO a través del vector de interrupciones, y el SO se encarga de terminar el programa de forma anormal, exhibiendo el mensaje de error apropiado y produciendo un vuelco de memoria.

2.5.1 Operación en modo dual

Se requiere protección para cualquier recurso compartido. Para ello el hardware debe distinguir como mínimo entre dos modos de ejecución, *modo de usuario* y *modo privilegiado*. Se le agrega al hardware del computador un BIT, llamado *BIT de modo*, para indicar en que modo se está usando.
En el momento de arrancar el sistema, el hardware inicia en modo privilegiado, luego se carga el sistema operativo y el modo pasa a ser usuario. Cada vez que ocurre una interrupción, el hardware pasa a modo privilegiado, y vuelve a cambiar a modo usuario antes de transferir el control a un programa de usuario.

Hay algunas instrucciones que podrían causar daño, y por ello solo se pueden usar en modo privilegiado. Si se intenta ejecutar una en modo usuario, el hardware la detecta, efectúa una interrupción y pasa el control al SO.

MS-DOS, no fue diseñado para modo dual (la arquitectura Intel 8088, no lo poseía), por lo tanto un programa de usuario fuera de control, puede borrar el sistema operativo escribiendo datos encima de él. A partir de la CPU 80486, implementaron el modo dual, por ello Windows NT (y otros), protegen el sistema contra estos errores.

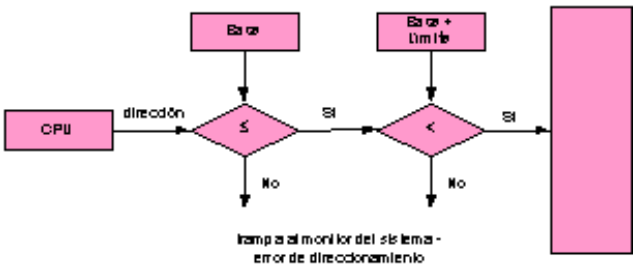
2.5.2 Protección de E/S

Para evitar que un usuario realice E/S no válida, todas las instrucciones de E/S son privilegiadas, los usuarios deben ceder el control al sistema operativo. Si puedo acceder a la memoria, al vector de interrupciones, y alterar la dirección de la interrupción que genera el hardware, puedo asumir el control del computador en modo privilegiado, para que esto no ocurra, también debo proteger la memoria.

2.5.3 Protección de la memoria

Debemos impedir que el vector de interrupción sea modificado por un programa de usuario, al igual que las rutinas de servicio de interrupciones del SO. Esta protección debe correr por cuenta del hardware.

Podemos separar el espacio de memoria de cada programa, determinando el espacio de direcciones, al cual el programa puede acceder, y proteger el resto de la memoria que no está en ese espacio. Se almacena en registros, el rango de memoria privilegiada, y se efectúa interrupción si cualquier programa en modo usuario, intenta acceder a esa zona de memoria.



Los registros de control son:

- **Registro base:** Contiene la dirección de memoria física, válida más pequeña.
- **Registro límite:** Contiene el tamaño del intervalo.

El hardware de la CPU, compara todas las direcciones generadas por el usuario, con estos registros, pasando el control al SO, si se produce algún error. El SO tiene acceso irrestricto a la memoria, pudiendo cargar los programas usuario en cualquier lado, y pudiendo expulsar esos programas en caso de error.

2.5.4 Protección de la CPU

Debemos impedir que un programa de usuario se atasque en un loop y nunca devuelva el control al sistema operativo. Para ello, existe un *timer*, que interrumpe al computador después de un período determinado, y cede el control al SO, que puede cerrar el programa o cederle más tiempo. Las instrucciones que modifican el funcionamiento del timer, son privilegiadas.

Otra implementación de los timers es para sistemas de tiempo compartido, dándole un tiempo determinado para los programas de cada usuario; o para el cálculo de la hora actual.

2.6 Arquitectura general de los sistemas

Dado que las instrucciones de E/S son privilegiadas, y solo el SO puede ejecutarlas, los programas de usuario, deben hacer un pedido al SO, para que realice la E/S en su nombre. Esta solicitud se llama *llamada al sistema*. El hardware trata esta interrupción como una interrupción de software, y pasa al modo privilegiado, entregándole el control al SO. Si la solicitud es válida, el SO efectúa la E/S solicitada, y devuelve el control al usuario, pasando nuevamente a modo usuario.
Otra instrucción como **halt** (parar), es privilegiada, ya que un programa de usuario no debe parar el computador. Las instrucciones para activar y desactivar el sistema de interrupciones también son privilegiadas, al igual que la instrucción que cambia del modo usuario al modo monitor. Al ser privilegiadas, solo las realiza el SO.

3 Estructuras del sistema operativo

Un SO crea el entorno en el que se ejecutan los programas. Los SO se pueden estudiar, examinando los servicios que proporciona, estudiando la interfaz con usuarios y programadores, o desglosándolo en sus componentes y viendo las interconexiones.

3.1 Componentes del sistema

Se divide en partes pequeñas, cada una bien delineada con entradas, salidas y funciones.

3.1.1 Gestión de procesos

Un proceso es una porción de programa en ejecución, el cual necesita tiempo de la CPU, memoria, archivos y dispositivos E/S. Esos recursos se le otorgan cuando se crea, o en el momento de ejecución.

Un programa es una entidad *pasiva* en cambio un proceso es una entidad *activa*. La ejecución de un proceso se debe realizar en secuencia, ejecutando una instrucción tras otra, hasta que el proceso termina. Dos procesos asociados al mismo programa se consideran como dos secuencias de ejecución individuales. Además, en un instante dado, se ejecuta como máximo una instrucción del proceso.

Un proceso es la unidad de trabajo de un sistema. El sistema consiste en una colección de procesos de dos tipos:

- **Del SO:** Ejecutan código del sistema
- **Del usuario:** Ejecutan código del usuario

Todos estos procesos se pueden ejecutar de forma concurrente, multiplexando la CPU entre ellos.

El SO respecto a gestión de procesos se encarga de:

- Crear y eliminar procesos, tanto de usuario como de sistema
- Suspender y reanudar procesos
- Proveer mecanismos para sincronizar procesos
- Proveer mecanismos para comunicar procesos
- Proveer mecanismos para manejar bloqueos mutuos

3.1.2 Gestión de memoria principal

La memoria principal es una matriz de palabras o bytes, que contiene datos a los cuales se puede acceder rápidamente y son compartidos por la CPU y los dispositivos E/S. Es el único dispositivo de almacenamiento grande que la CPU puede direccionar y acceder directamente.
Para ejecutar un programa, es preciso cargarlo en memoria y transformar sus direcciones relativas en absolutas.
Para mejorar el grado de utilización del CPU, y la rapidez con la que el computador responde al usuario, es necesario tener varios programas en memoria. Hay muchos esquemas de gestión de memoria, que dependen principalmente del hardware.
El SO se encarga de:

- Saber que partes de la memoria se están usando, y quien las usa
- Decidir que procesos cargar en memoria cuando se disponga de espacio
- Asignar y liberar memoria según se necesite

3.1.3 Gestión de archivos

Los sistemas almacenan información en medios físicos como cintas o discos, los cuales tienen sus propias características y organización, y son controlados por un dispositivo.
El SO define una perspectiva lógica de almacenamiento, independiente del dispositivo (abstrayendo el hardware), y es el archivo, pero debe establecer una correspondencia, entre el archivo y el medio físico.
Los archivos normalmente se organizan en directorios para facilitar su uso, y si pueden ser accedidos por varios usuarios a la vez, el SO debe llevar un control.

El SO se encarga de:

- Crear y eliminar archivos
- Crear y eliminar directorios
- Proveer las primitivas de manipulación de archivos y directorios
- Establecer la correspondencia de archivos con el almacenamiento secundario
- Resguardarlos en un medio de almacenamiento no volátil

3.1.4 Gestión del sistema de E/S

Uno de los objetivos del SO es ocultar las peculiaridades de dispositivos de hardware específicos para que el usuario no las perciba. Solo el driver del dispositivo conoce las peculiaridades, y el sistema proporciona una interfaz general con los controladores de dispositivos.

3.1.5 Gestión de almacenamiento secundario

Como la memoria principal es pequeña y volátil, para dar cabida a todos los datos y programas, el almacenamiento secundario debe respaldar la memoria principal. Casi todos los programas se guardan en un disco, hasta que se cargan en memoria, por eso es importante una correcta administración del almacenamiento.
El SO se encarga de:

- Administración de espacio libre
- Asignación del almacenamiento
- Planificación del disco

El almacenamiento secundario es frecuente así que su uso debe ser eficiente. La velocidad del computador podría depender del subsistema de disco y los algoritmos que lo manipulan.

3.1.6 Trabajo con redes

En una red, hay una colección de procesadores, que no comparten memoria, ni periféricos ni reloj, los cuales se comunican a través de una red de comunicaciones (sistema *distribuido*).
La red puede estar total o parcialmente conectada y su diseño debe considerar los problemas de contención y seguridad.
El acceso a los recursos compartidos, permite acelerar cálculos, y los SO gestionan esa red.

3.1.7 Sistema de protección

Si un SO tiene múltiples usuarios, y permite la ejecución concurrente de procesos, debe proteger cada proceso de las actividades de los demás. Debe asegurar que solo aquellos procesos que obtuvieron autorización debida del SO, puedan operar con archivos, memoria, CPU o demás recursos.

3.1.8 Sistema de interpretación de órdenes

Uno de los programas más importantes de un sistema operativo es el *intérprete de comandos*, que es la interfaz entre el usuario y el sistema. Algunos SO lo incluyen en el núcleo, otros como MS-DOS y UNIX, lo tratan como un programa especial que se ejecuta cuando un usuario entra en el sistema.
Muchas de las órdenes al SO vienen de enunciados de control. El programa que interpreta estos comandos es llamado "*shell*", y su función es la de obtener la siguiente orden y ejecutarla.
Los shell se diferencian por la "amabilidad" hacia el usuario. Pueden ser con sistema de ventanas como el de Macintosh o Windows, u otros más potentes, complejos y difíciles, donde las órdenes se teclean y exhiben en pantalla, como los de MS-DOS y UNIX.
Las órdenes mismas sirven para crear y administrar procesos, manejar la E/S, administrar el almacenamiento secundario,

acceder al sistema de archivos, etc.

3.2 Servicios del sistema operativo

El sistema operativo proporciona servicios a los programas y a los usuarios de esos programas. Estos varían de un SO a otro, pero hay clases comunes.

- **Ejecución de programas:** El sistema debe poder cargar un programa en la memoria y ejecutarlo. El programa debe poder terminar su ejecución, normal o anormalmente (indicando el error).
- **Operaciones de E/S:** Un programa en ejecución podría requerir E/S, implicando el uso de un archivo o dispositivo E/S. Por cuestiones de seguridad y eficiencia, los usuarios casi nunca pueden controlar los dispositivos de E/S directamente, por lo tanto el sistema debe incluir un mecanismo para realizarlo.
- **Manipulación del sistema de archivos:** Los programas necesitan leer, escribir, crear y eliminar archivos.
- **Comunicaciones:** Los procesos necesitan intercambiar información con otro. Puede hacerse usando memoria compartida, o transfiriendo mensajes, y hay dos formas de comunicación:
  - Ocurre entre procesos que se ejecutan en el mismo computador.
  - Implica procesos que se ejecutan en computadores distintos conectados en una red.
- **Detección de errores:** Pueden ocurrir errores en el hardware, en los dispositivos E/S o en programas de usuario. Para cada tipo de error, el sistema operativo debe emprender la acción apropiada para asegurar un funcionamiento correcto y consistente del sistema de computación.

Hay otro grupo de funciones que aseguran el funcionamiento eficiente del sistema:

- **Asignación de recursos:** Si hay varios usuarios o varios trabajos ejecutándose al mismo tiempo, es preciso asignar recursos a cada uno. Algunos tienen código de asignación especial, como memoria principal y almacenamiento de archivos. Otros tienen código de solicitud y liberación mucho más general, como los dispositivos de E/S.
- **Contabilización:** Es importante saber que usuarios están usando recursos, cuantos recursos usan, y de que tipo.
- **Protección:** Los dueños de una información quieren controlar su uso. Aseguro que todos los accesos a los recursos del sistema sean controlados. Obligo a que cada usuario del sistema deba identificarse, por lo regular con una contraseña antes de poder acceder a los recursos. Esto sirve para defender los dispositivos de E/S externos contra intentos de acceso no autorizado, y registrar todas las conexiones.

3.3 Llamadas al sistema

Son la interfaz entre un proceso y el sistema operativo, y están generalmente disponibles como instrucciones en lenguaje ensamblador, aunque algunos sistemas permiten emitir llamadas al sistema desde un programa escrito en lenguaje de alto nivel. Estas llamadas al sistema se pueden agrupar en cinco categorías principales.

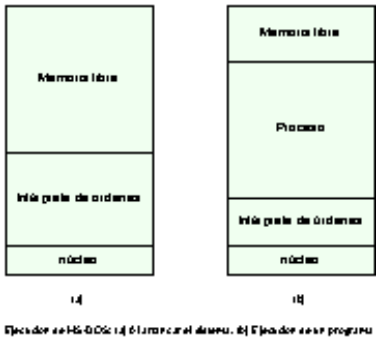
3.3.1 Control de procesos y trabajos

- **Fin, abortar:** Un programa en ejecución debe poder detenerse ya sea normalmente (end) como anormalmente (abort). Si el programa se topa con algún problema hace un vuelco de memoria y se genera un mensaje de error.
- **Cargar, ejecutar:** Un proceso o trabajo que ejecuta un programa, podría querer cargar y ejecutar otro programa. Estas funciones permiten al intérprete, ejecutar un programa cuando se le solicita, y luego devolver el control al programa que se estaba ejecutando anteriormente.
- **Crear proceso, terminar proceso:** Siguiendo el caso anterior, si el programa llamado, continúa ejecutándose en forma concurrente con el programa que lo llamó, hemos creado un nuevo trabajo, o proceso. Si vemos que no se necesita más, podemos terminar el proceso.
- **Obtener atributos de proceso, establecer atributos de proceso:** Si quiero determinar y reestablecer los atributos de un

proceso como su prioridad, o su tiempo de ejecución máximo, obtengo y establezco los atributos del proceso.

- **Esperar un lapso de tiempo:** Después de crear procesos, es posible que tengamos que esperar a que terminen de ejecutarse.
- **Esperar suceso, indicar la ocurrencia de suceso:** Es más probable que deseemos esperar que ocurra un suceso específico, por lo tanto el proceso deberá generar una señal cuando ese suceso haya ocurrido.
- **Asignar y liberar memoria:** Ayuda a depurar programas.

MS-DOS:



Es un sistema monotarea, que no crea un proceso nuevo para ejecutar un programa. Carga en programa en memoria escribiendo sobre casi todo su propio código para dar al programa el máximo de memoria posible. Cuando el programa termina, la pequeña porción que quedó del intérprete de órdenes, reanuda su ejecución, volviéndose a cargar en memoria.
El sistema ofrece un método para una ejecución concurrente limitada. Un programa, puede salir con la llamada al sistema “terminar y seguir residente”, la cual hace que DOS reserve el espacio ocupado por el programa, de modo que no se sobrescriba.

UNIX:

Es un sistema multitareas. Cuando un usuario ingresa al sistema, se ejecuta el shell preferido por el usuario. La interfaz es similar a la de MS-DOS, pero como es un sistema multitareas, el shell puede seguirse ejecutando mientras hay un programa en memoria. Por lo tanto el usuario está en condiciones de pedir al shell que ejecute otros programas.

3.3.2 Manipulación de archivos

- **Crear archivo, eliminar archivo:** Ambas llamadas necesitan el nombre del archivo, y tal vez alguno de sus atributos.
- **Abrir, cerrar:** Una vez creado, se necesita abrirlo para usarlo, y cerrarlo cuando no lo necesitemos más.
- **Leer, escribir, reposicionar:** Mientras el archivo está abierto, podemos leer, escribir y reposicionarnos (saltar al fin de archivo, volver al principio).
- **Obtener atributos de archivo, establecer atributos de archivo:** Necesitamos obtener los valores de algunos de los atributos del archivo, o cambiarlos.

3.3.3 Gestión de dispositivos

Un programa podría requerir recursos adicionales durante su ejecución, como más memoria, acceso a archivos, etc. El SO debe administrarlos, para ver si están ocupados, etc.

- **Solicitar dispositivo, liberar dispositivo:** Cuando hay múltiples usuarios, debo solicitar los dispositivos, a fin de obtener su uso exclusivo, cuando no lo necesito más, debo liberarlo.
- **Leer, escribir, reposicionar:** Trabajo con el dispositivo.

La gestión de dispositivo es similar a la de los archivos.

3.3.4 Mantenimiento de información

Hay llamadas al sistema para obtener la hora y fecha actuales, devolver información de sistema, como número de usuarios actuales, espacio libre en el disco, etc.
El sistema operativo mantiene información acerca de todos sus procesos, y hay llamadas al sistema para obtenerla, y restablecer

la información.

3.3.5 Comunicación

Hay dos modelos de comunicación comunes.

- **Transferencia de mensajes:** La información se intercambia por medio de un recurso de comunicación entre procesos provisto por el SO. Para hacerlo se debe conocer el nombre del computador en la red (host name).
  - *Abrir conexión, cerrar conexión:* Se piden para establecer la conexión, y para que se establezca, el host debe aceptar la conexión.
  - *Leer mensaje, escribir mensaje:* Se usan para comunicarse.
- **Memoria compartida:** Los procesos utilizan llamadas al sistema para obtener acceso a regiones privadas de memoria de otros procesos, estas zonas no están controladas por el SO.

Ambos métodos son comunes en los sistemas operativos, y algunos, implementan los dos. La transferencia de mensajes es útil cuando se desean intercambiar pocos datos, evitando conflictos. La memoria compartida es más rápida, pero hay problemas en las áreas de protección y sincronización.

3.4 Programas del sistema

Los programas del sistema se pueden dividir en varias categorías:

- **Manipulación de archivos:** Crean, eliminan, copian, cambian de nombre, imprimen, vacían, listan y manipulan archivos y directorios.
- **Información de estado:** Piden al sistema la fecha, hora, cantidad de memoria, espacio en disco disponible, numero de usuarios, etc.
- **Modificación de archivos:** Puede contarse con varios editores de texto para crear y modificar el contenido de archivos.
- **Apoyo a lenguajes de programación:** Proporcionan al usuario compiladores, ensambladores, intérpretes de lenguajes. Actualmente muchos de esos programas, se venden por separado.

- **Carga y ejecución de programas:** Puede incluir cargadores absolutos, relocalizables, editores de enlaces y cargadores de superposiciones.
- **Comunicaciones:** Crean conexiones virtuales entre procesos, usuarios y sistemas distintos; permiten a los usuarios enviar mensajes a otros, correo electrónico, transferencia de archivos e incluso usar computadores remotos como si fueran locales.

La mayor parte de los SO cuentan con programas que ayudan a resolver problemas comunes, como navegadores Web, procesadores de textos, hojas de cálculo, sistemas de bases de datos, paquetes de graficación, y juegos. Estos programas son conocidos como programas de aplicación.

El programa más importante para un SO es el *intérprete de órdenes* que tiene como función, obtener y ejecutar la siguiente orden especificada por el usuario.

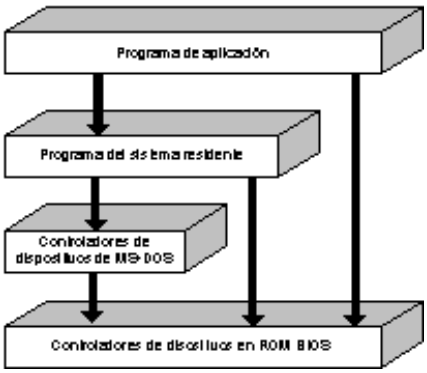
Ubicación del código para ejecutar órdenes:

- **En el intérprete:** La cantidad de órdenes determina el tamaño del intérprete.
- **Programas del sistema:** El intérprete no entiende la orden, sino que la usa para identificar que archivo del sistema debe cargar en memoria y ejecutar. De esta forma, se pueden añadir nuevas órdenes creando archivos con el nombre correspondiente.

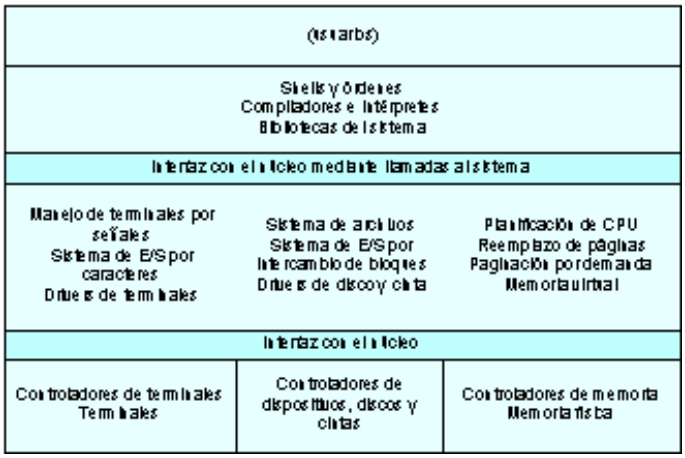
3.5 Estructura del sistema

El sistema está dividido en componentes pequeños, donde cada uno de los módulos debe ser una porción bien definida del sistema. Estos componentes se conectan entre sí, formando un núcleo.

3.5.1 Estructura simple



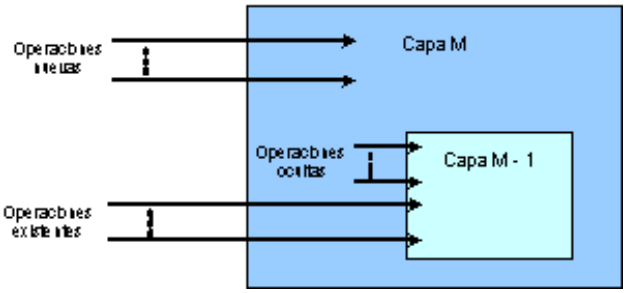
Hay muchos sistemas comerciales que no tienen una estructura bien definida. Es común que tales sistemas hayan sido pequeños en sus comienzos y luego hayan crecido más allá de su alcance original. MS-DOS es un ejemplo.
Las interfaces y niveles de funcionalidad no están bien separados. Por ejemplo, los programas de aplicación pueden acceder a las rutinas de E/S básicas para escribir directamente en pantalla y disco, dejando el sistema a merced de programas mal intencionados, cayendo el sistema cuando un programa falla.
MS-DOS también estaba limitado por el hardware de su época (8088) que no poseía ni modo dual, ni protección por hardware.



UNIX inicialmente también estuvo limitado por la funcionalidad del hardware. Consiste en dos partes separables, el núcleo y los programas del sistema. El núcleo se divide a su vez en una serie de interfaces y controladores de dispositivos que se han agregado y ampliado a medida que UNIX ha evolucionado. Hay una enorme cantidad de funcionalidad combinada en un solo nivel.
Las llamadas al sistema definen la *interfaz del programador* con UNIX, los programas del sistema, definen la *interfaz con el usuario*. Estas dos definen el contexto que el núcleo debe apoyar.

3.5.2 Enfoque por capas

Si se cuenta con el apoyo apropiado de hardware, los sistemas operativos se pueden dividir en fragmentos más pequeños y adecuados que los originales como en el caso de MS-DOS y UNIX. Así el SO puede mantener un control mucho mayor sobre el computador y las aplicaciones que lo usan.
Una de las formas de modularizar, consiste en dividir el sistema en varias capas, cada una construida sobre las capas inferiores. La capa 0 es el hardware, y la n (la más alta) es la interfaz con el usuario.



Una capa es una implementación de un objeto abstracto, que es el encapsulamiento de datos y operaciones que manipulan esos datos.

La capa M consiste en estructuras de datos y rutinas que las capas de nivel superior pueden invocar. A su vez M puede invocar funciones de capas de niveles más bajos. Cada capa se implementa utilizando solo operaciones provistas por capas del nivel inferior.

Ventajas:

- Al utilizar solo operaciones implementadas en el nivel inferior, se simplifica la depuración y verificación del sistema.
- Se logra abstracción, ya que no sabemos como la capa inferior implementó la operación, sino que solo la usamos. Cada

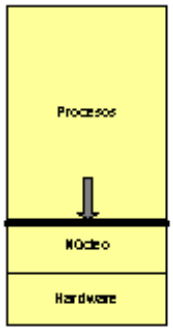
capa oculta las estructuras de datos, operaciones y hardware, de las capas superiores.

Desventajas:

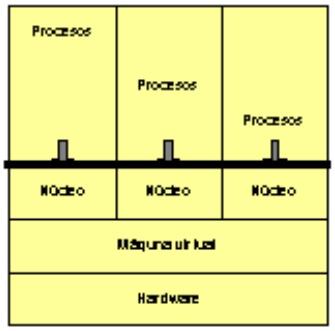
- El principal problema es la definición apropiada, puesto que una capa solo puede usar las capas de nivel más bajo, y la planificación debe ser muy cuidadosa.
- El sistema por capas tiende a ser menos eficiente, ya que una rutina invoca a otra y así sucesivamente, creando un gasto extra.

3.6 Máquinas virtuales

Un sistema de computador se compone de capas, donde la más baja es el hardware. El núcleo que se ejecuta en el siguiente nivel, utiliza las instrucciones del hardware para crear un conjunto de llamadas al sistema, que las capas exteriores pueden usar.



Máquina no virtual



Máquina virtual

Usando planificación de CPU y técnicas de memoria virtual, un sistema operativo puede crear la ilusión de que múltiples procesos se ejecutan cada uno en su propio procesador con su propia memoria. Los recursos del computador físico se comparten para crear las máquinas virtuales. Se puede usar planificación del CPU para compartirla entre los usuarios.
Un problema con la máquina virtual, tiene que ver con los sistemas de disco. Si una máquina tiene tres discos, y se quieren crear 7 máquinas virtuales, no puedo asignarle un disco a cada una. La solución es ofrecer discos virtuales a cada una de ellas, donde la suma del tamaño de los discos virtuales, es menor que el espacio físico de los discos.

3.6.1 Implementación

El concepto de máquina virtual, es simple, pero difícil de implementar. La máquina virtual en sí, solo puede ejecutarse en modo usuario, pero el software virtual, puede hacerlo en modo monitor, puesto que es el sistema operativo. En consecuencia, debemos tener un modo usuario y monitor virtual, que se ejecutan en modo de usuario físico, por lo tanto, todas las acciones que permiten el paso de modo usuario a modo monitor en una máquina real, también deben hacerlo en una virtual.
Es posible de hacer, pero la diferencia, es el tiempo, ya que es necesario simular todas las instrucciones, y se ejecutan más lentamente.

3.6.2 Beneficios

La máquina virtual está completamente aislada de las demás, por lo tanto la protección de los recursos del sistema, es total. Los recursos no se comparten directamente, y se puede definir una red de máquinas virtuales cada una de las cuales puede enviar información a través de una red de comunicaciones virtual.
Las máquinas virtuales se están poniendo de moda para resolver los problemas de compatibilidad de sistemas. Sirve para que programas hechos para un sistema determinado puedan ser ejecutados en otro sistema no compatible.

3.6.3 Java

El compilador de Java genera *códigos de bytes* como salida, que se ejecutan en la *Máquina Virtual Java (JVM)*. Cada sistema tiene su JVM, que controla seguridad y confiabilidad del programa. Por lo tanto se logran programas seguros y transportables, que a pesar que no son tan rápidos como los compilados para hardware nativo, si son más eficientes, y tienen ventajas sobre ellos.

3.7 Diseño e implementación de sistemas

3.7.1 Objetivos de diseño

El primer problema es definir los objetivos y especificaciones del sistema. Seleccionar el hardware y el tipo de procesamiento que se implementará (por lotes, tiempo compartido, etc.).

Más allá de eso, hay que especificar los requisitos, como:

- **Metas del usuario:** Que sea cómodo de usar, fácil de aprender, confiable, seguro y rápido.
- **Metas del sistema:** Fácil de diseñar, implementar y mantener; flexible, confiable, libre de errores y eficiente.

Las especificaciones anteriores no son útiles para el diseño, ya que no explican como debe hacerse. No existe una solución única. La amplia gama de sistemas demuestra que diferentes requisitos dan pie a diferentes entornos. A pesar de la variedad, hay principios generales que se han sugerido.

3.7.2 Mecanismos y políticas

Los mecanismos determinan cómo se hace algo, y las políticas, qué se hará. La separación de política y mecanismo es importante para la flexibilidad. Las políticas pueden cambiar de un lugar a otro, o de un momento a otro.
Los sistemas operativos basados en micro núcleo llevan la separación entre mecanismo y política al extremo, implementando un conjunto básico de bloques de construcción primitivos, los cuales son independientes de la política, y permiten agregar mecanismos y políticas mas avanzados mediante módulos del núcleo creados por el usuario.
En el otro extremo esta el sistema de la Apple Macintosh, donde el mecanismo y la política están incorporados en el código del sistema, para que tenga aspecto y forma de uso global. Todas las aplicaciones tienen interfaces similares, porque la interfaz está incluida en el núcleo.

3.7.3 Implementación

Una vez diseñado el sistema, hay que implementarlo. Tradicionalmente los sistemas se escribían en lenguaje ensamblador. Ahora es posible escribirlos en lenguajes de alto nivel.
La ventaja de usar un lenguaje de alto nivel, es que el código se puede escribir más rápidamente, es más compacto, y más fácil de entender y depurar; además los adelantos en los compiladores, pueden mejorar el sistema, con solo recompilar el código, además es más fácil de transportar (trasladar a otro hardware) si está escrito en un lenguaje de alto nivel.
Las desventajas de usar lenguajes de alto nivel, son una menor velocidad, y mayores necesidades de almacenamiento, aunque esto va mejorando, a medida que los compiladores avanzan, generando código cada vez más compacto y eficiente.
Una vez que se escribió un sistema, es posible identificar las rutinas que son cuellos de botella y sustituirlas por equivalentes en lenguaje ensamblador. Para identificar estos cuellos de botella, se añaden códigos que calculan y exhiben medidas del comportamiento del sistema.

3.8 Generación de sistemas

Los sistemas operativos deben ser diseñados, para que se ejecuten en cualquier máquina de una clase dada, en diversos sitios con diferentes configuraciones de periféricos. Para eso es necesario configurar el sistema para cada computador específico. El proceso se conoce como *generación de sistemas*.
El programa SYSGEN (system generation) pide al operador del sistema, información sobre la configuración del hardware, o sondea el hardware directamente. Hay que determinar los siguientes tipos de información:

- **CPU:** Qué CPU se usará, que opciones están instaladas (aritmética de punto flotante, etc.), y en caso de sistemas con múltiples CPU, hay que describir a cada uno.
- **Memoria:** Cuanta memoria posee. Algunos sistemas determinan el valor por si solos, haciendo referencia a una posición tras otra, hasta que se genera un fallo de “dirección no válida”.
- **Dispositivos:** Que dispositivo se posee, como direccionarlo, el número de interrupción, el tipo y modelo y características especiales que tenga.
- **Parámetros del sistema:** Que opciones o valores de parámetros se usarán en el sistema operativo. Estas opciones pueden ser cantidad de buffers, tamaño, número de procesos simultáneos, etc.

Una vez determinada esta información, puede servir para modificar una copia del código fuente del sistema.
En un nivel, menos “personalizado”, se pueden crear bibliotecas con configuraciones, que se anexan al sistema operativo, logrando que el sistema tenga controladores de dispositivos para todos los dispositivos de E/S conocidos, y solo los necesarios se enlazarán al sistema.

Una vez generado el sistema operativo, debe ponerse a disposición del hardware. El procedimiento de iniciar un computador, cargando el núcleo se denomina *arranque del sistema*. En la mayor parte de los computadores, hay un pequeño fragmento de código, almacenado en ROM, llamado *programa de arranque*. Este código puede localizar el núcleo, cargarlo en memoria principal y comenzar su ejecución.

4 Procesos

Podemos considerar un proceso como un programa en ejecución. Necesita ciertos recursos, como tiempo de CPU, memoria, archivos y dispositivos E/S, los cuales son asignados cuando se crea, o durante su ejecución.
El proceso es la unidad de trabajo en la mayor parte de los sistemas. Un sistema entonces consiste en una colección de procesos (procesos de usuario y procesos de sistema).

Los primeros sistemas solo permitían la ejecución de un programa a la vez, el cual asumía el control total del sistema y tenía acceso a todos sus recursos. Los sistemas actuales permiten cargar múltiples programas en memoria y ejecutarlos en forma concurrente. Para ello se debe controlar los distintos programas, dando pie al concepto de proceso.
Un sistema consiste en una colección de procesos: Procesos del sistema operativo, que ejecutan código del sistema, y procesos del usuario, que ejecutan código del usuario. Todos esos procesos podrían ejecutarse de forma concurrente, multiplexando la CPU entre ellos.

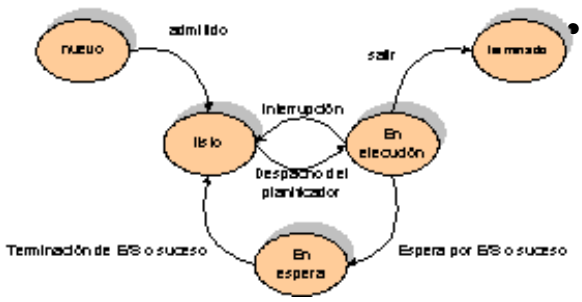
4.1 El concepto de proceso

4.1.1 El proceso

Un proceso es un programa en ejecución, la cual debe proceder de manera secuencial. Un proceso además del código del programa, incluye la actividad actual, representada por el valor del *contador de programa*, y el valor de los registros de la CPU. Generalmente también incluye al *stack* del proceso, que contiene datos temporales, y una sección de datos que contiene variables globales.
Un programa, es una entidad pasiva, como un archivo en disco, mientras que un proceso es una entidad activa.
Aunque podría haber dos procesos asociados al mismo programa, de todas maneras se consideran como dos secuencias de ejecución distintas.

4.1.2 Estado de un proceso

A medida que un proceso se ejecuta, cambia de estado, pudiendo estar en uno de los siguientes:



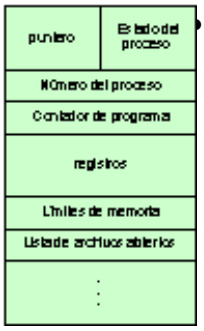
- **Nuevo:** El proceso se está creando
- **En ejecución:** Se están ejecutando instrucciones
- **En espera:** Está esperando que ocurra algún suceso (terminación del E/S o recepción de señal)
- **Listo:** Está esperando que se le asigne a un procesador
- **Terminado:** Terminó su ejecución

Solo un proceso puede estar ejecutándose en un instante dado, pero muchos procesos pueden estar listos y esperando.

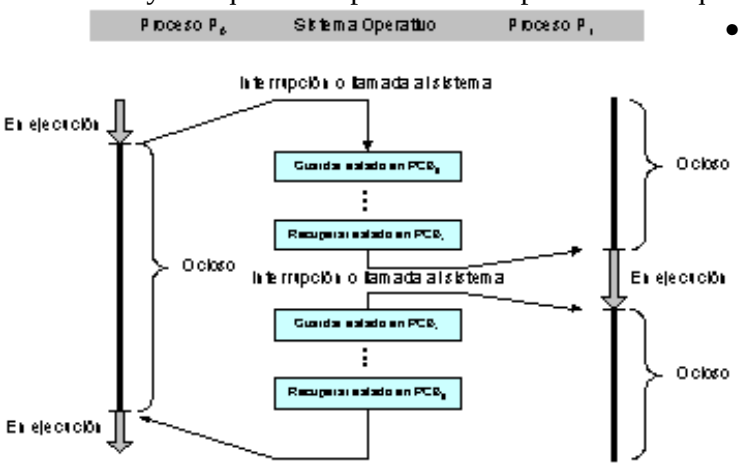
4.1.3 Bloque de control de proceso

Cada proceso se representa en el sistema operativo con un bloque de control de proceso.

- **Estado del proceso:** El estado puede ser nuevo, listo, en ejecución, en espera, detenido, etc.
- **Contador de programa:** Indica la dirección de la siguiente instrucción que se ejecutará para este proceso.



- **Registros de CPU:** El número y tipo de los registros varía dependiendo de la arquitectura del computador. Los registros incluyen acumuladores, registros índices, punteros de pila y registros de propósito general, etc. Esta información se debe guardar cuando ocurre una interrupción, para que el proceso pueda después continuar correctamente.
- **Información de planificación de CPU:** Incluye la prioridad del proceso, punteros a colas de planificación y cualquier otro parámetro de planificación que haya.

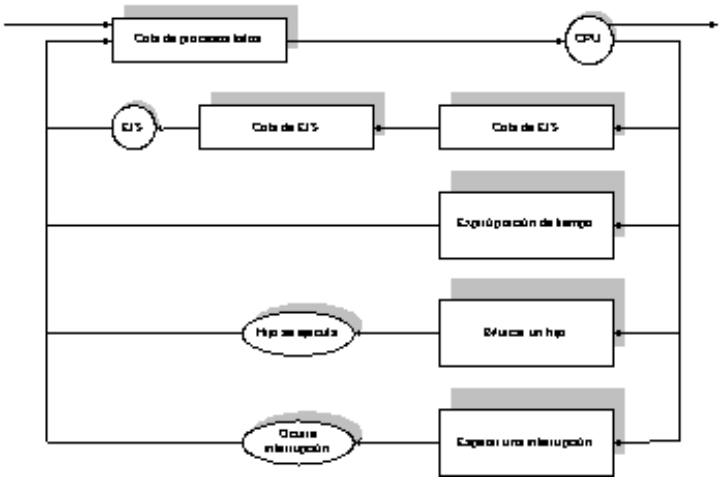


- **Información de gestión de memoria:** Incluye el valor de los registros base y límite, las tablas de páginas o segmentos, etc.
- **Información contable:** Cantidad de tiempo de CPU y tiempo real consumida, límites de tiempo, números de cuenta, números de trabajo o proceso, etc.
- **Información de estado de E/S:** Incluye la lista de dispositivos de E/S asignadas a este proceso, una lista de archivos abierta, etc.

4.2 Planificación de procesos

El objetivo de la multiprogramación es tener algún proceso de ejecución en todo momento, a fin de aprovechar al máximo la CPU. El objetivo es conmutar la CPU entre procesos con tal frecuencia que los usuarios puedan interactuar con cada programa durante su ejecución. En el caso de un único CPU, solo se ejecuta un proceso por vez.

4.2.1 Colas de planificación



Conforme los procesos ingresan al sistema, se colocan en una *cola de trabajos*. Los procesos que están en la memoria principal, y esperando para ejecutarse, se mantienen en una cola, llamada *cola de procesos listos*.
Hay otras colas en el sistema, como por ejemplo la lista de procesos que están esperando un dispositivo de E/S (un disco, o unidad de cinta, etc.), se le denomina *cola de dispositivo*, y cada dispositivo tiene su propia cola.
Un proceso nuevo se coloca inicialmente en la cola de procesos listos, donde espera hasta que se le escoge para ejecutarse y recibe la CPU. Una vez que se está ejecutando, puede ocurrir:

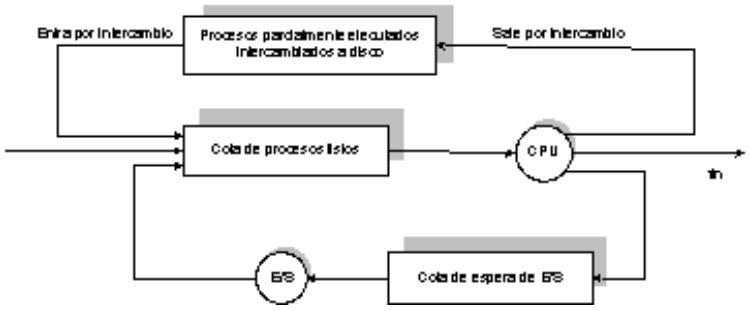
- El proceso podría emitir una solicitud E/S, y entonces colocarse en una cola de E/S.
- El proceso podría crear un subproceso y esperar a que termine.
- El proceso podría ser desalojado por la fuerza de la CPU, como resultado de una interrupción, y ser colocado otra vez en la cola de procesos listos.

En los dos primeros casos, el proceso pasa del estado de espera al estado listo, y se vuelve a colocar en la cola de los procesos listos.

4.2.2 Planificadores

Un proceso migra de una cola de planificación a otra durante toda su existencia, por lo tanto, el sistema operativo, debe seleccionar procesos de estas colas de alguna manera. El planificador apropiado se encarga de la selección.
Tipos:

- **Corto plazo:** Llamado planificador de la CPU. Escoge entre los procesos que están listos para ejecutarse y asigna a la CPU, uno de ellos. Actúa más frecuentemente que los otros, por lo tanto debe ser muy rápido.
- **Largo plazo:** Llamado planificador de trabajos. Escoge procesos de reserva y los carga en memoria para que se ejecuten. Se ejecuta con menos frecuencia, y controla el grado de multiprogramación (número de procesos en memoria). Este planificador, puede tardar más en elegir el proceso a ejecutar, por su baja frecuencia, y por ello hace una selección cuidadosa. Hay procesos que están limitados por la E/S, o el CPU, y es importante que el planificador los mezcle para aprovechar los recursos. Es probable que los sistemas de tiempo compartido, no tengan planificador a largo plazo, y se limiten a colocar todos los procesos nuevos en la memoria para el planificador a corto plazo.
    - Un proceso limitado por la E/S es uno que dedica la mayor parte del tiempo a operaciones E/S, y menos a realizar cálculos.
    - Un proceso limitado por CPU es lo contrario.
- **Mediano plazo:** Son provechosos para sacar procesos de memoria, a fin de reducir el grado de multiprogramación. En algún momento posterior, el proceso se reintroduce en la memoria y continúa la ejecución desde donde quedó. Este esquema se denomina intercambio (swapping). Es útil para cuando tengo memoria ocupada y necesito liberarla.



4.2.3 Conmutación de contexto

El cambio de la CPU a otro proceso, requiere guardar el estado del proceso anterior, y cargar el estado guardado del nuevo proceso. Esta tarea se denomina *conmutación de contexto*, y es un gasto extra, porque el sistema no realiza ningún trabajo útil durante la conmutación.
Cuanto más complejo sea el sistema operativo, más trabajo hay que efectuar en la conmutación. Esta conmutación se ha convertido en un cuello de botella, tan importante que se están empleando estructuras nuevas (hilos) para evitarla.

4.3 Operaciones con procesos

Los procesos se pueden ejecutar de forma concurrente, y se deben crear y eliminar dinámicamente. El SO debe contar con mecanismos para ello.

4.3.1 Creación de procesos

Un proceso puede crear varios procesos nuevos a través de una llamada al sistema “crear proceso”, durante su ejecución. El proceso creador es el proceso *padre*, y los nuevos procesos son los *hijos*. Estos hijos pueden crear nuevos procesos, generando un *árbol* de procesos.
Un proceso necesita recursos (CPU, memoria, archivos, E/S) para efectuar su tarea. Cuando crea un subproceso, éste último podría obtener los recursos directamente del SO, pero podría estar restringido a un subconjunto de los recursos del padre. El padre quizás deba dividir sus recursos entre sus hijos, o tal vez varios hijos pueden compartir recursos. Esta restricción de

recursos impide que cualquier proceso sobrecargue el sistema creando demasiados subprocesos.
Cuando un proceso crea otro nuevo, hay dos posibilidades de ejecución:

- El padre sigue ejecutándose de forma concurrente con sus hijos
- El padre espera hasta que algunos de sus hijos, o todos hayan terminado.

También hay posibilidades en términos del espacio de direcciones del nuevo proceso:

- El proceso hijo es un duplicado del padre.
- Se carga un programa en el proceso hijo.

UNIX

Cada proceso se identifica con su *identificador de proceso (PID)*. Se crea un nuevo proceso con la llamada al sistema **fork**, y el nuevo proceso es una copia del espacio de direcciones del proceso original, permitiendo una fácil comunicación entre el padre y el hijo. Ambos procesos continúan ejecutando la instrucción siguiente al fork, con la diferencia de que el código de retorno que recibe el hijo del fork es cero, y el del padre, es el PID del hijo.
El padre puede crear más hijos, o si no tiene nada para hacer, ejecuta **wait** sacándose a si mismo de la cola de procesos listos, hasta que el hijo termine.

4.3.2 Terminación de procesos

Un proceso acaba cuando termina de ejecutar su último enunciado y pide la salida al SO mediante la llamada al sistema **exit**. En ese momento, puede devolver datos al proceso padre, y el SO libera todos los recursos del proceso, incluidos memoria física y virtual, archivos abiertos y buffers de E/S.
Un proceso puede causar la terminación de otro con una llamada al sistema apropiada como “abortar”. Normalmente solo el padre del proceso puede hacerlo, de otro modo, los usuarios podrían matar los trabajos de otro usuario. Las razones para terminar la ejecución de un hijo son:

- El hijo excedió los recursos asignados. Para saberlo, el padre debe contar con un mecanismo para inspeccionar el estado de los hijos.
- La tarea asignada al hijo ya no es necesaria
- El padre va a salir, y el SO no permite que el hijo continúe sin su padre. El SO se encarga de la terminación en cascada (eliminar todos los hijos).

4.4 Procesos cooperativos

Tipos de procesos concurrentes:

- **Independiente:** No puede ser afectado ni afectar los demás procesos. No comparte datos con ningún otro proceso.
- **Cooperativo:** Puede afectar o ser afectado por otros procesos. Comparte datos con ellos.

Las razones para permitir la cooperación son:

- **Compartir información:** Varios usuarios podrían estar interesados en el mismo elemento de información (Ej.: un archivo).
- **Aceleración de los cálculos:** Divido las tareas en subtareas, y las ejecuto concurrentemente. Esta aceleración sólo puede lograrse en sistemas multiprocesador.
- **Modularidad:** Se puede seguir un patrón modular, dividiendo las funciones del sistema en procesos individuales.
- **Comodidad:** Un mismo usuario podría tener muchas tareas que realizar en un momento dado. Ej. Podría estar imprimiendo, compilando y editando al mismo tiempo.

Esta ejecución concurrente con cooperación, requiere mecanismos que permitan comunicación y sincronización entre procesos.

Ejemplo:

Un proceso *productor* produce información que es consumida por un proceso *consumidor*. Para que los procesos puedan ejecutarse de forma concurrente, es preciso contar con un buffer de elementos, que el productor pueda llenar, y el consumidor vaciar. Un productor puede producir un elemento mientras el consumidor consume otro. El productor y el consumidor deben estar sincronizados para que el consumidor no trate de consumir un elemento que todavía no se ha producido.

- No solo hay que controlar que el consumidor no consuma, si el buffer está vacío, sino también, que al no existir un buffer ilimitado, el productor debe esperar si el buffer está lleno.

4.5 Hilos (Threads)

Un proceso se define por los recursos que usa y por el lugar donde se ejecuta, pero hay muchos casos en los que sería útil compartir los recursos y acceder en forma concurrente.

4.5.1 Estructura de los hilos

Un hilo (proceso ligero) es una unidad básica de utilización de la CPU, y consiste en un contador de programa, un juego de registros y un espacio de pila. El hilo comparte con sus hilos pares la sección de código, la sección de datos y los recursos del sistema operativo como archivos abiertos.
El compartir hace que la conmutación de la CPU entre hilos sea mucho más económica que las conmutaciones de contexto, ya que no hay que realizar operaciones relacionadas con la gestión de memoria. Pero al igual que cualquier entorno de proceso paralelo, puede traer problemas de control de concurrencia.
Algunos sistemas implementan *hilos en el nivel de usuario*, en bibliotecas de usuario, por lo tanto la conmutación de hilos no necesita invocar al sistema operativo ni causar interrupciones para pasar al núcleo.
La organización en procesos es útil cuando los trabajos que realizan no tienen relación entre ellos, ya que no se comparte ni el contador de programa, ni el registro de pila, ni el espacio de direcciones.
Los hilos operan en muchos sentidos, de la misma forma que los procesos. Pueden estar en varios estados (listo, bloqueado, en ejecución o terminado). Pero a diferencia de los procesos, los hilos no son independientes entre si y dado que todos pueden acceder a todas las direcciones de la tarea, un hilo puede leer o escribir sobre la pila de otro.
La estructura no ofrece protección entre hilos, sin embargo, esa protección no debería ser necesaria, puesto que el dueño de la tarea con múltiples hilos, es un único usuario.
El núcleo del sistema puede apoyar los hilos, contando con llamadas al sistema similares a las usadas con procesos, otra alternativa es apoyarlos mediante bibliotecas de usuario. La conmutación entre hilos en nivel usuario, es mucho más rápida, sin embargo, las llamadas al sistema operativo pueden hacer que todo el proceso espere, porque el núcleo solo planifica procesos.
Algunos sistemas adoptan un enfoque híbrido en el que se implementan tanto hilos en el nivel de usuario, como apoyados por el núcleo (Ej: Solaris 2).

4.5.2 Ejemplo: Solaris 2

Apoya hilos en los dos niveles, multiprocesamiento simétrico y planificación en tiempo real. Tiene un nivel intermedio de hilos, llamados *procesos ligeros*.
Los recursos necesarios para los hilos son:

- **Núcleo:** Solo tiene una estructura de datos y una pila. La conmutación es rápida porque no se accede a la memoria.
- **Procesos ligeros:** Tiene un bloque de control de procesos con datos de registros, información de contabilidad e información de memoria. La conmutación es lenta.
- **Usuario:** Solo tiene una pila y un contador. El núcleo no interviene por lo tanto la conmutación es rápida.

4.6 Comunicación entre procesos

Los procesos cooperativos pueden comunicarse en un entorno de memoria compartida, teniendo una reserva de buffers en común, implementados por el programador de la aplicación. Otra forma es que el SO proporcione los medios de comunicación, a través de un mecanismo de comunicación y sincronización entre procesos (IPC).
La mejor forma de proveer la comunicación es mediante un sistema de mensajes, aunque también puede hacerse compartiendo la memoria, o los dos simultáneamente.

4.6.1 Estructura básica

La función de un sistema de mensajes es permitir a los procesos comunicarse entre si sin tener que recurrir a variables compartidas, ofreciendo como mínimo las funciones **send** y **recieve**. Los mensajes pueden ser de tamaño fijo o variable. Si es fijo, la implementación es sencilla, pero dificulta la tarea de programación, los variables requieren una implementación más compleja y simplifican la programación.
Para que los procesos P y Q se envíen mensajes, debe existir un *enlace de comunicación* entre ellos, el cual se puede implementar de diversas maneras.
Un enlace es *unidireccional*, sólo si cada proceso conectado al enlace puede enviar o recibir, pero no ambas cosas, y cada enlace tiene por lo menos un proceso receptor conectado a él. Las formas de implementar lógicamente un enlace, y las operaciones de enviar/recibir son:

- Comunicación directa o indirecta
- Comunicación simétrica o asimétrica
- Uso de buffers automático o explícito
- Envío por copia o envío por referencia
- Mensajes de tamaño fijo o variable

4.6.2 Asignación de nombres

Para comunicarse, deben tener una forma de referirse unos a otros. Se puede usar comunicación directa o indirecta.

4.6.2.1 Comunicación directa

Cada proceso debe nombrar explícitamente el destinatario o remitente de la comunicación. Las primitivas de enviar y recibir son:

Enviar (P, mensaje)	Enviar mensaje al proceso P
Recibir (Q, mensaje)	Recibir un mensaje del proceso Q

Las propiedades del enlace son:

- Se establece automáticamente un enlace entre cada par de procesos que desean comunicarse, y los procesos solo necesitan conocer la identidad del otro para comunicarse.
- Un enlace se asocia a exactamente dos procesos.
- Entre cada par de procesos solo existe un enlace.
- El enlace puede ser unidireccional, pero suele ser bidireccional.

El problema es la limitada modularidad de las definiciones de procesos. Hay que hallar todas las referencias al nombre antiguo a fin de poder cambiarlas, lo cual no es deseable.

4.6.2.2 Comunicación indirecta

Los mensajes se envían y reciben a través de buzones. Un buzón es un objeto en el que los procesos pueden colocar y sacar mensajes. Cada buzón tiene una identificación única, y un proceso se puede comunicar con otro a través de varios buzones distintos, pero se pueden comunicar solo si comparten el buzón. Las primitivas enviar y recibir se definen de la forma:

Enviar (A, mensaje)	Enviar el mensaje al buzón A
Recibir (A, mensaje)	Recibir un mensaje del buzón A

Las propiedades del enlace son:

- Se establece un enlace entre un par de procesos sólo si tienen un buzón compartido.
- Un enlace puede estar asociado a más de dos procesos.
- Entre cada par de procesos puede haber varios enlaces distintos, cada uno de los cuales corresponderá a un buzón.
- Los enlaces pueden ser unidireccionales o bidireccionales.

Tipos de buzón:

- ***Buzón de proceso:*** Existe un propietario que solo puede recibir mensajes a través de ese buzón, y un usuario, que solo puede enviar. Cuando el proceso que posee el buzón termina, el buzón desaparece, notificando que ya no existe a los procesos que intentan utilizarlo.
- ***Buzón del sistema:*** Tiene existencia propia, es independiente y no está unido a ningún proceso específico. El proceso que crea el buzón, es su dueño, y es el único que puede recibir mensajes a través de él, pero la propiedad y privilegio, pueden ser trasferidas a otros procesos por medio de llamadas al sistema, logrando múltiples receptores para el buzón.

4.6.3 Uso de buffers

Las formas de implementar la cola de mensajes, dependiendo de la capacidad del enlace son:

- **Capacidad cero:** Tiene como longitud máxima cero, el enlace no puede tener mensajes esperando en él. El emisor debe esperar hasta que el destinatario reciba el mensaje. Los dos procesos deben sincronizarse para realizar la transferencia. Es llamado también, sistema de mensajes sin buffer.
- **Capacidad limitada:** La cola tiene una longitud finita n, por lo tanto n mensajes pueden residir en ella. Si la cola no está llena, el emisor puede continuar su ejecución sin esperar, sino debe esperar hasta que haya espacio libre en la cola.
- **Capacidad ilimitada:** La cola tiene una longitud potencialmente infinita, entran cualquier cantidad de mensajes, y el emisor nunca espera.

Casos especiales:

- **El proceso que envía un mensaje nunca espera:** Si el receptor no recibió el mensaje, antes de que el emisor le envíe otro, el primero se perderá. Los procesos necesitan sincronizarse para asegurar de que no se pierdan los mensajes, y de que el emisor y el receptor no manipulen el buffer simultáneamente.
- **El proceso que envía un mensaje espera hasta recibir una respuesta:** El emisor se bloquea al enviar el mensaje, hasta recibir una respuesta del receptor.

4.6.4 Condiciones de excepción

Los sistemas de mensajes son útiles en los entornos distribuidos, donde los procesos podrían residir en diferentes sitios, pero la probabilidad de que ocurra un error durante la comunicación en este entorno, es mucho mayor que en una sola máquina.
En una máquina, se implementan como memoria compartida, y si ocurre una falla, todo el sistema falla. En un entorno distribuido, el fallo de un sitio no causa necesariamente un fallo en todo el sistema.
Cuando ocurre un fallo, el sistema debe intentar recuperarse del error, manejando algunas de las condiciones de excepción.

4.6.4.1 El proceso termina

El emisor o el receptor podrían terminar antes de que se procese un mensaje, dejando mensajes que nunca se recibirán o procesos esperando mensajes que nunca se enviarán.

- Un proceso receptor P, podría esperar el mensaje de Q que ya terminó. Si no se toman medidas, P esperaría eternamente. En este caso, el sistema podría terminar P, o notificarle que Q terminó.
- El proceso P podría enviarle un mensaje a Q que ya terminó. Con el esquema de buffers, no hay problema, P sigue funcionando. Si P necesita saber que Q procesó el mensaje y no se usan buffers, P se bloqueará eternamente. El sistema podría avisarle que Q ya terminó, o podría terminar P.

4.6.4.2 Mensajes perdidos

Un mensaje podría perderse en la red de comunicación a causa de un fallo de hardware o de la línea de comunicaciones. Hay tres métodos para enfrentar ese suceso:

- El sistema debe detectarlo y retransmitir el mensaje.
- El proceso emisor debe detectarlo y retransmitir el mensaje si desea hacerlo.
- El sistema debe detectarlo, y avisarle al emisor que el mensaje se perdió. El emisor decide que hacer.

No siempre es necesario detectar mensajes perdidos. El usuario debe especificar si quiere hacerlo.
La forma más común de detectar errores es empleando *tiempos límite*. Cuando se envía un mensaje, se espera una respuesta. Si la respuesta no llega antes del plazo establecido, el proceso emisor supone que el mensaje se perdió, y lo reenvía. Puede suceder que no se haya perdido en realidad, y solo haya tardado más de lo esperado, en este caso podrían existir múltiples copias del mismo mensaje, por lo tanto se debe distinguir si un mensaje es repetido.

4.6.4.3 Mensajes alterados

El mensaje podría llegar, pero podría sufrir alteraciones en el camino, en este caso el SO debe reenviar el mensaje. Para detectar los errores comúnmente se usan códigos de verificación de errores.

4.6.5 Un ejemplo: Mach

Casi todas las llamadas al sistema, y toda la transferencia de información entre tareas, se realiza con mensajes, enviándolos o recibiéndolos a través de buzones (puertos).
La tarea que crea el buzón, es la dueña, y solo ella puede recibir de ese buzón, aunque puede ceder los derechos a otras tareas.
Todos los mensajes tienen igual prioridad, y se despachan en orden de llegada (FIFO).
Los mensajes consisten en una cabecera de longitud fija, seguida de datos de longitud variable. La cabecera guarda la longitud del mensaje y dos nombres de buzón (emisor y receptor).

5 Planificación de la CPU

La planificación de la CPU es la base de los SO multiprogramados. Al conmutar la CPU entre procesos, el SO puede hacer más productivo el computador.

5.1 Conceptos básicos

El objetivo es tener algún proceso en ejecución en todo momento maximizando el uso del CPU. Un proceso se ejecuta hasta que tiene que esperar, casi siempre para atender una solicitud E/S. Mientras el proceso espera, el SO le quita la CPU y se la da a otro proceso.

Casi todos los recursos del computador se planifican antes de usarse, principalmente la CPU.

5.1.1 Ciclo de ráfagas de CPU y E/S

Los procesos en ejecución alternan entre dos estados (ciclos de CPU y espera por E/S), terminando en una ráfaga de CPU, con solicitud al sistema, para terminar la ejecución. Los programas limitados por E/S tienen muchas ráfagas de CPU cortas, en cambio los no limitados, pueden tener ráfagas de CPU más largas.

5.1.2 Planificador de CPU

Siempre que la CPU esta ociosa, se debe escoger un proceso, de la cola de procesos listos. Esta selección corre por cuenta del planificador a corto plazo. La cola de procesos no es FIFO, sino que es una cola de prioridad.

5.1.3 Planificación expropiativa

Las decisiones de planificación de CPU se toman en las cuatro situaciones siguientes:

1. Cuando un proceso pasa del estado de ejecución al estado en espera (solicitud E/S, o espera a que termine un proceso hijo).
2. Cuando un proceso pasa del estado en ejecución al estado listo (ocurrencia de una interrupción).
3. Cuando un proceso pasa del estado en espera al estado listo (terminación de E/S).
4. Cuando un proceso termina.

En los casos 1 y 4, se escoge un proceso nuevo. El esquema de planificación es no expropiativo. En los casos 2 y 3, que si hay opciones, el esquema es expropiativo.

- **No expropiativo:** Una vez que la CPU se ha asignado a un proceso, éste la conserva hasta que la cede, porque terminó, o porque paso al estado de espera. Se utiliza en algunas plataformas que no requieren hardware especial, ya que el proceso expropiativo debe contar con un temporalizador.
- **Expropiativo:** Implica un costo, ya que se debe implementar mecanismos para coordinar el acceso a datos compartidos, como por ejemplo, cuando un proceso esta escribiendo datos, y otro intenta leerlos al mismo tiempo.

5.1.4 Despachador

Es el módulo que cede el control de la CPU al proceso seleccionado por el planificador. Esta función implica:

- Cambiar el contexto
- Cambiar a modo usuario
- Saltar el punto apropiado del programa de usuario, para reiniciar ese programa

Debe ser lo más rápido posible, porque se invoca en cada conmutación. El tiempo de demora, se denomina *latencia del despachador*.

5.2 Criterios de planificación

Hay diferentes algoritmos de planificación, los cuales deben ser escogidos con cuidado. Los criterios de comparación empleados para la selección son:

- **Utilización de la CPU:** Queremos mantener la CPU tan ocupada como se pueda.
- **Rendimiento:** Si la CPU está ocupada ejecutando procesos, se está efectuando trabajo. Una medida del trabajo, es el número de procesos completados por unidad de tiempo, y es llamada *rendimiento*.
- **Tiempo de retorno:** Es importante el tiempo que tarda la ejecución de un proceso, o sea, desde que empieza, hasta que

termina, contando todos los tiempos de espera.

- **Tiempo de espera:** No afecta el tiempo que pasa realizando E/S, sino el que pasa esperando en la cola de procesos listos. El tiempo de espera, es la suma de los períodos que el proceso pasa esperando en dicha cola.
- **Tiempo de respuesta:** Es el tiempo que transcurre entre la presentación de una solicitud, y la presentación de la primera respuesta. Generalmente está limitado por la velocidad del dispositivo de salida.

Si queremos garantizar que todos los usuarios reciban buen servicio, debemos minimizar el tiempo de respuesta.

5.3 Algoritmos de planificación

Deciden que proceso de la cola de procesos listos, debe recibir la CPU.

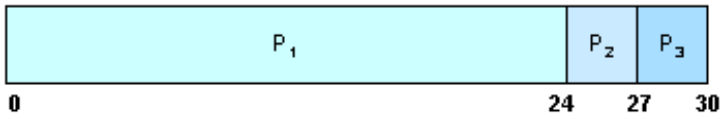
5.3.1 Planificación de “servicio por orden de llegada”

Se lo conoce por FCFS (first come, first served). El proceso que primero solicita la CPU, primero la recibe. Se implementa con una cola FIFO. El problema es que el tiempo de espera promedio suele ser muy largo.

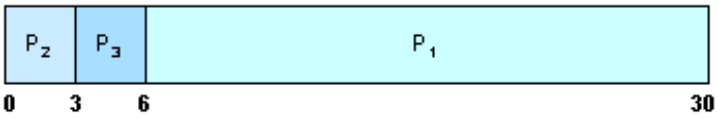
Ejemplo:

P_1	24
P_2	3
P_3	3

Si los procesos se atienden por orden de llegada:



El tiempo de espera, es 0 para P_1, 24 para P_2 y 27 para P_3, sumando en total (0 + 24 + 27) / 3 = 17 milisegundos. En cambio si cambio el orden a:



El tiempo de espera promedio es ahora (0 + 3 + 6) / 3 = 3.

Por lo visto, el tiempo promedio de espera casi nunca es mínimo,
en estos casos se puede presentar el “efecto convoy”, ya que todos los procesos deben esperara a que un solo proceso grande desocupe la CPU, aprovechando de forma menos eficiente la CPU y los dispositivos, que si se dejara paso a procesos más cortos. Este algoritmo es especialmente problemático en los sistemas de tiempo compartido, donde es importante que los usuarios reciban el tiempo de CPU a intervalos regulares.

5.3.2 Planificación de “primero el trabajo más corto”

Siempre se elige el proceso que tiene la ráfaga de CPU más corta, eligiendo de forma similar a la anterior, a los procesos con ráfagas de igual duración (orden de llegada). Es conocido como shortest job first (SJF).
Este algoritmo de planificación es óptimo, ya que el tiempo de espera promedio es mínimo. Lo realmente difícil, es conocer la duración de la siguiente solicitud del CPU. Se puede usar como duración límite, el tiempo que el usuario especifica, por lo tanto los usuarios deben calcular los tiempos de forma precisa, para que su solicitud sea atendida lo más rápido posible (si ponen un valor demasiado bajo, causarían error de tiempo excedido, teniendo que presentar el trabajo nuevamente).
No hay forma de conocer la duración de la siguiente ráfaga. Una forma, es tratar de aproximar esta planificación. No conocemos la duración, pero quizás podamos predecir su valor.

5.3.3 Planificación por prioridad

Se asocia una prioridad a cada proceso, y la CPU se asigna al proceso con mayor prioridad. Cuanto más larga es la ráfaga de CPU, más baja es la prioridad.
Las prioridades pueden ser:

- **Interna:** Se utiliza una o más cantidades medibles para calcular la prioridad. Por ejemplo: límites de tiempo, necesidad

de memoria, etc.
- **Externa:** Se fija empleando criterios externos al SO, como importancia del proceso, los fondos necesarios para poder usar el computador, etc.

Cuando un proceso llega a la cola de procesos listos, su prioridad se compara con la del proceso que se está ejecutando. La planificación por prioridad puede ser:

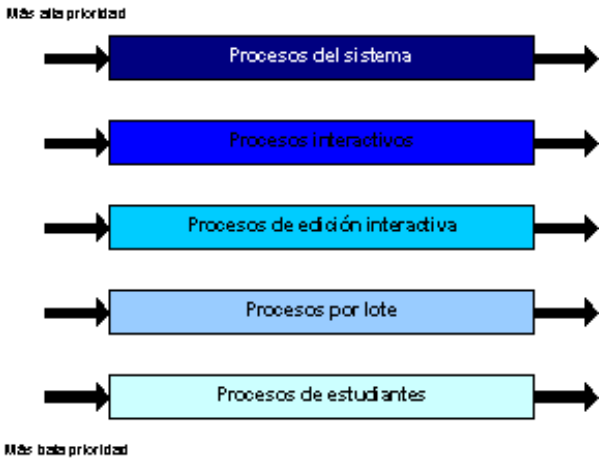
- **Expropiativa:** El algoritmo de planificación se apropia de la CPU si la prioridad del proceso recién llegado es mayor que la del que se está ejecutando.
- **No expropiativo:** Coloca el nuevo proceso a la cabeza de la cola de procesos.

Un problema de estos algoritmos de planificación es el “Bloqueo indefinido”. Podrían dejar algunos procesos de baja prioridad esperando indefinidamente la CPU, sobretodo si en el sistema hay un flujo cargado de procesos.
Una solución a este problema es el “envejecimiento”, que consiste en aumentar gradualmente la prioridad de los procesos que esperan mucho tiempo en el sistema.

5.3.4 Planificación por turno circular

Se diseñó para los sistemas de tiempo compartido y es similar a la planificación por orden de llegada, pero existe la expropiación para conmutar entre procesos. La cola de procesos es circular. Se define una unidad de tiempo pequeña y el planificador recorre la cola, asignando la CPU a cada proceso por una unidad de tiempo. El algoritmo es conocido como Round Robin (RR).
Si el proceso tiene una ráfaga de CPU menor a la unidad de tiempo, él mismo libera la CPU, sino, el temporalizador genera una interrupción al SO, se conmuta el contexto, y se pone el proceso al final de la cola.
El tiempo de espera promedio en estos casos es muy grande, además de que si los proceso tardan más que la unidad de tiempo, se genera un gasto extra por cada conmutación de contexto.

5.3.5 Planificación con colas de múltiples niveles



Es útil cuando es fácil clasificar los procesos en diferentes grupos. Se divide la cola de procesos listos en varias colas distintas, y asigna los procesos a la cola correspondiente dependiendo de sus propiedades.
Cada cola tiene su algoritmo de planificación, y hay planificación entre todas las colas, la cual es generalmente expropiativa de prioridades fijas; además cada cola tiene prioridad absoluta sobre las colas de más baja prioridad.
Otra posibilidad es dividir el tiempo entre las colas, donde cada una obtiene cierta porción de tiempo de la CPU, que reparte entre sus procesos.

5.3.6 Planificación con colas de múltiples niveles y realimentación

En el caso anterior, los procesos se asignan a las colas al ingresar al sistema. Los procesos no se mueven de una cola a otra, teniendo como ventaja que le gasto extra por planificación es bajo, y como desventaja que es inflexible.
En este caso, se permite pasar de una cola a otra. La idea es separar los procesos con diferentes características en cuanto a sus ráfagas de CPU. Si gasta demasiado tiempo, se le baja la prioridad, y si está demasiado tiempo en la cola, se le sube.
El planificador está definido por los siguientes parámetros:

- Número de colas
- Algoritmo de planificación para cada cola
- Método empleado para decidir el aumento de prioridad de un proceso

- Método empleado para decidir la disminución de prioridad de un proceso
- Método empleado para decidir en que cola ingresar el proceso cuando necesite servicio
- Este es el algoritmo más general, ya que se puede adaptar al SO.

5.4 Planificación de múltiples procesadores

Si hay varias CPU, la planificación se hace más compleja, y no hay tampoco una solución óptima.
En un sistema multiprocesador homogéneo, puede haber limitaciones para la planificación. Si el sistema tiene un dispositivo conectado al bus privado de un procesador, los procesos que intentan acceder a un mismo dispositivo deben planificarse para acceder a ese procesador.
Si se establece una cola para cada procesador, podrían haber algunos sobrecargados, y otros ociosos; para evitar esto, se usa una cola común de procesos listos, asignando a ellos cualquier procesador disponible.

Hay dos formas de planificación:

- Cada procesador se auto planifica (examina la cola y elige el proceso)
- Se nombra a un procesador como planificador de los demás, evitando conflictos para acceder a la cola, como en el caso anterior (Master – slave)

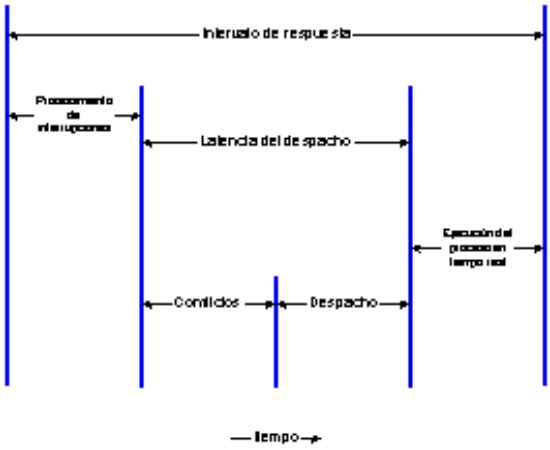
Otros sistemas tienen un procesador que se encarga de las E/S, actividades del sistema y planificación mientras los demás procesadores ejecutan código de usuario. Este es *multiprocesamiento asimétrico.*

5.5 Planificación en tiempo real

La computación en tiempo real se divide en dos tipos:

Sistemas de tiempo real duro: Deben completar una tarea crítica dentro de un lapso de tiempo garantizado. Los procesos se presentan junto con una declaración del tiempo necesario, y el planificador los rechaza si no va a poder cumplir con ese tiempo. Esto se denomina *reservación de recursos*.
Una garantía así es imposible en un sistema con almacenamiento secundario o memoria virtual, por lo tanto los sistemas de tiempo real duro consisten en software especial, ejecutado en hardware dedicado al proceso crítico.

Sistemas de tiempo real blando: Los procesos críticos tienen mayor prioridad que los demás. Esto podría dar pie a una asignación de recursos no equitativa o inanición. Este sistema puede apoyar multimedia y gráficos de alta velocidad, que no serían posibles en otro sistema.
La implementación debe ser cuidadosa con el planificador, y debe ser por prioridad, asignando la más alta a los procesos críticos, la cual no se puede degradar. La latencia del despacho debe ser pequeña.



La primera propiedad es simple de cumplir, en cambio la segunda es más complicada. Para hacerlo necesitamos que las llamadas al SO, sean expropiativas.
La latencia del despacho tiene tres componentes:

1. Desalojo de cualquier proceso que se esté ejecutando en el núcleo.
2. Liberación por parte de los procesos de menor prioridad, de los recursos que el proceso de alta prioridad necesita.
3. Conmutación de contexto actual con el de alta prioridad.

5.6 Evaluación de algoritmos

Para seleccionar un algoritmo hay que ver la importancia relativa de las siguientes medidas:

- Maximizar el aprovechamiento de la CPU, tomando como tiempo de respuesta máximo el de un segundo.
- Maximizar el rendimiento de modo que el tiempo de retorno promedio sea linealmente proporcional al tiempo de ejecución total.

5.6.1 Modelado determinista

La evaluación analítica utiliza el algoritmo dado y la carga de trabajo del sistema para producir un número que califica el desempeño del algoritmo para esa carga de trabajo.
Un tipo de evaluación analítica es el *modelo determinista* que toma una carga de trabajo predeterminada específica y define el desempeño del algoritmo para esa carga. Este modelo es sencillo, rápido, y da números exactos, aunque también requiere números exactos como entrada, por lo tanto es demasiado específico para ser útil.

5.6.2 Modelos de colas

No hay un conjunto estático de procesos para usar con el modelo determinista, lo que se puede medir, es la distribución de las ráfagas de CPU y E/S. Así mismo, debe darse la distribución de los tiempos en que los procesos llegan al sistema, y a partir de las dos distribuciones, se calcula el rendimiento promedio.
La CPU tiene cola de procesos listos, y la E/S, colas de dispositivos. Si se conoce la frecuencia de llegada, y la rapidez del servicio, se puede calcular la longitud promedio de las colas, y el tiempo de espera promedio. Este estudio se denomina *análisis de redes de colas*.
Este análisis sirve para comparar algoritmos de planificación, pero tiene limitaciones, ya que la clase de algoritmos y distribuciones que pueden manejarse es limitada, y hay que hacer supuestos que podrían ser no validos.

5.6.3 Simulaciones

Se programa un modelo del sistema. Existe un reloj, que cuando se incrementa su valor, el simulador modifica el estado del sistema reflejando las actividades de los dispositivos, proceso y planificador, recopilando datos estadísticos.

5.6.4 Implementación

Las simulaciones tienen exactitud limitada. La forma exacta de evaluar un algoritmo es codificarlo, colocarlo en el SO y ver como funciona.
El principal problema es el costo del enfoque, además de que los programas nuevos se adaptan al algoritmo. Ej: Si se da prioridad a los procesos cortos, los usuarios dividen sus procesos largos en grupos de procesos pequeños, para ejecutarlos más rápidamente.

6 Sincronización de procesos

Un proceso cooperativo puede afectar o ser afectado por los demás. Podrían compartir un espacio de direcciones lógicas (código y datos), o solo datos a través de archivos. En el primer caso, es a través de hilos, lo que puede dar pie a inconsistencias.

Ejemplo de concurrencia

```
FACTORIAL (N: Integer)
BEGIN
        COBEGIN
                a = Semifact (N, ½)
                b = Semifact (N/(2 - 1), 1)
        COEND
        fact = a * b
END
```

Todo lo que está entre COBEGIN y COEND se puede ejecutar de forma simultánea.

Ejemplo de no concurrencia

```
J = 10
COBEGIN
        printj
        j= 1000
COEND
printj
```

La primera impresión, no se si es 10 o 1000, todo depende de que instrucción se ejecute primero.
La segunda impresión siempre es 1000, porque hasta que no se ejecute todo lo que está entre el COBEGIN y COEND no se sigue.

6.1 Antecedentes

Cuando compartimos a través de un buffer, hay que controlar si se vacía o llena. Podemos añadir un contador que indica la cantidad de elementos del buffer.
El problema es que como los procesos que leen y escriben son concurrentes, pueden aparecer problemas al actualizar el valor de esa variable, dependiendo de cómo se ejecuten las instrucciones de máquina. Debemos lograr que solo un proceso a la vez trabaje con el contador.

Ejemplo:

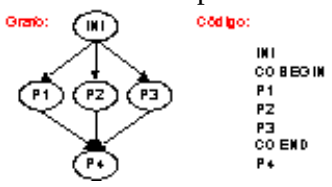
```
1 Reg1 = cont          4 Reg2 = cont
2 Reg1 = Reg1 +1       5 Reg2 = Reg2 - 1
3 cont = Reg1          6 cont = Reg2
```

La variable contador puede guardar distintos valores. Si el contador al comienzo es 6, y ejecuto las instrucciones:

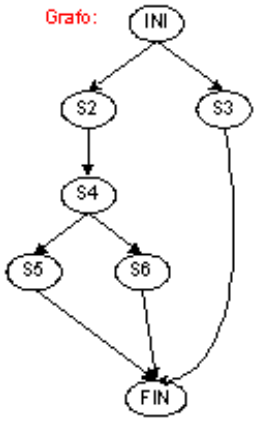
- 1 – 4 – 2 – 5 – 3 – 6: El contador queda en 5
- 1 – 2 – 3 – 4 – 5 – 6: El contador queda en 6

Grafo de precedencia

Es un grafo acíclico, dirigido, cuyos nodos corresponden a las sentencias, y las flechas a la precedencia.



Ejemplo:

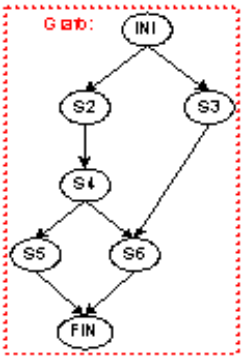


Código 1:

```
INI
COBEGIN
        P
        S3
COEND
FIN

P:
BEGIN
        S2
        S4
        COBEGIN
                S5
                S6
        COEND
END
```

Código 2:

```
INI
COBEGIN
        BEGIN
                S2
                S4
                COBEGIN
                        S5
                        S6
                COEND
        END
        S3
COEND
FIN
```

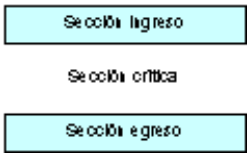


Ejemplo:

En este caso no hay forma de representar el grafo con COBEGIN y COEND, sin incorporar otras primitivas.

6.2 El problema de la sección crítica

Consideramos el sistema con n procesos $\{P_0, \dots, P_n\}$, donde cada cual tiene un segmento de código, llamado *sección crítica*, en el que el proceso podría estar modificando variables comunes, etc.



La idea es que mientras un proceso se está ejecutando en sección crítica, ningún otro puede estar en otra sección crítica; solo ejecutando otras secciones.
Así la ejecución de secciones críticas, es *mutuamente exclusiva*. Cada proceso debe solicitar permiso para ingresar en su sección crítica y debe indicar cuando salió de ella.

Hay cuatro requisitos:

- **Mutua exclusión:** Si el proceso P_i se ejecuta en su sección crítica, ningún otro proceso puede estarse ejecutando en su sección crítica.
- **Espera limitada:** Hay un límite, para el número de veces que se permite a otros procesos entrar en sus secciones críticas, luego de que un proceso pidió autorización de ingreso. No se permite la *posposición indefinida*.
- **Progreso:** Se deben cumplir las siguientes condiciones:
  - **No alternación:** No puedo obligar a que se ejecuten, primero uno, luego el otro, vuelvo al primero, etc.
  - **Sin puntos muertos (Deadlock):** A espera a B, y B espera a A, y los dos están bloqueados simultáneamente.

6.2.1 Soluciones para dos procesos

Para explicarlo mejor vamos a usar el problema de:
“Alicia y Bernardo, son vecinos, y tienen un patio en común. En ese patio quieren pasear sus perros, pero los dos juntos se pelean, así que tienen que turnarse”

6.2.1.1 Algoritmo 1

Los dos comparten una variable global turno, y dependiendo del valor de la variable, pasean al perro.

Se cumple la mutua exclusión, pero no la no alternación (progreso), ya que hasta que Bernardo no pasee el perro, Alicia no puede hacerlo.

```
PROCEDURE Alicia
        while true do
        {
                while turno <> 1 do;
                PASEO_PERRO
                turno = 2;
                OTRAS_TAREAS
        }
END

PROCEDURE Bernardo
        while true do
        {
                while turno <> 2 do;
                PASEO_PERRO
                turno = 1;
                OTRAS_TAREAS
        }
END

BEGIN
        turno = 1;
        COBEGIN
                Alicia
                Bernardo
        COEND
END
```

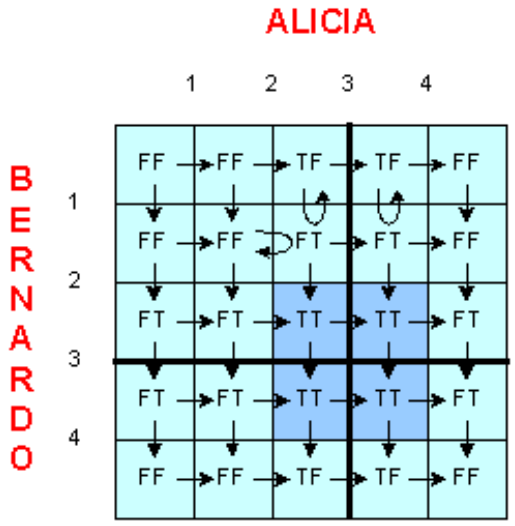
6.2.1.2 Algoritmo 2

Otra solución es colocar banderas al salir. Miro si el otro tiene la bandera colocada, y si no la tiene, salgo.

```
PROCEDURE Alicia
        while true do
        {
                1 - while flag_b do;
                2 - flag_a = true;
                3 - PASEO_PERRO
                4 - flag_a = false;
                OTRAS_TAREAS
        }
END

PROCEDURE Bernardo
        while true do
        {
                1 - while flag_a do;
                2 - flag_b = true;
                3 - PASEO_PERRO
                4 - flag_b = false;
                OTRAS_TAREAS
        }
END

BEGIN
        flag_a = false;
        flag_b = false;
        COBEGIN
                Alicia
                Bernardo
        COEND
END
```

Surge un problema, y es que puede suceder, que los dos evalúen las flags al mismo tiempo, dando true, y salgan los dos a pasear el perro.
No se satisface la condición de mutua exclusión.

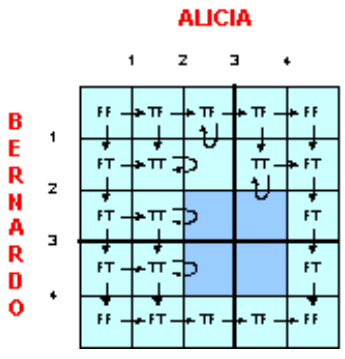
Para analizar la situación, se utiliza el método de entrelazado, que representa el estado de las banderas en cada una de las instrucciones.



Algoritmo 2 bis

Otra forma de hacerlo, es antes mirar la bandera del otro, levantar la de uno, mostrando la intención de pasear al perro, y luego miro si puedo.

```
PROCEDURE Alicia
        while true do
        {
                1 - flag_a = true;
                2 - while flag_b do;
                3 - PASEO_PERRO
                4 - flag_a = false;
                OTRAS_TAREAS
        }
END

PROCEDURE Bernardo
        while true do
        {
                1 - flag_b = true;
                2 - while flag_a do;
                3 - PASEO_PERRO
                4 - flag_b = false;
                OTRAS_TAREAS
        }
END

BEGIN
        flag_a = false;
        flag_b = false;
        COBEGIN
                Alicia
                Bernardo
        COEND
END
```



Resuelvo el problema de mutua exclusión, pero puedo llegar a DEADLOCK (Progreso)

6.2.1.3 Algoritmo 3

Si combinamos los dos primeros algoritmos, llegamos a la solución definitiva, llamada “Algoritmo de Becker”

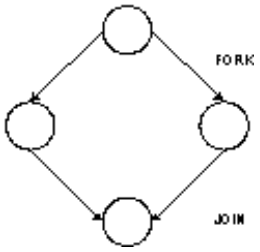
```
PROCEDURE Alicia
        while true do
        {
                flag_a = true;
                while flag_b do
                {
                        if (turno == 2)then
                        {
                                flag_a = false
                                while (turno == 2) do;
                                flag_a = true;
                        }
                }
                PASEO_PERRO
                turno = 2;
                flag_a = false;
                OTRAS_TAREAS
        }
END
```

```
PROCEDURE Bernardo
        while true do
        {
                flag_b = true;
                while flag_a do
                {
                        if (turno == 1)then
                        {
                                flag_b = false
                                while (turno == 1) do;
                                flag_b = true;
                        }
                }
                PASEO_PERRO
                turno = 1;
                flag_b = false;
                OTRAS_TAREAS
        }
END
```

```
BEGIN
        flag_a = false;
        flag_b = false;
        turno = 1;
        COBEGIN
                Alicia
                Bernardo
        COEND
END
```

La solución de Peterson soluciona igual que Becker, pero es más general, ya que esta solución es complicada para más de dos personas.

FORK – JOIN




```
----------
----------
----------
FORK ET1
----------
----------
----------
ET1
----------
----------
----------
```

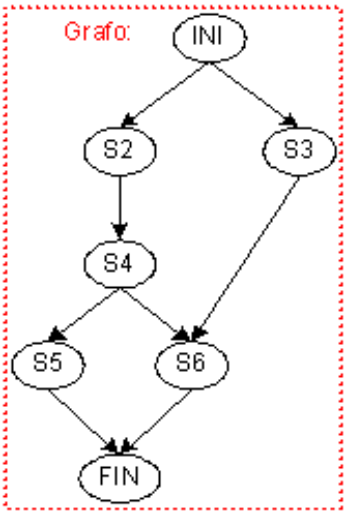
Sintaxis: FORK etiqueta JOIN m, etiqueta

Es como un GOTO, donde un proceso se ejecuta siguiente, y el otro después de la etiqueta. En el caso del JOIN, decremento m en uno, y si m = 0, salto a la etiqueta. Estas dos instrucciones, se ejecutan como instrucción única, de corrido.

Si hay n procesos concurrentes, el join mata uno a uno, y recién el último, pasa a ejecutar el proceso siguiente.

Ejemplo: El caso que no se podía solucionar con COBEGIN Y COEND

```
S1
cont1 = 2
cont2 = 2
```




```
FORK L1
S2
S4
FORK L2
S5
GOTO L7

L1 :
S3

L2:
JOIN cont1, L6
QUIT

L6:
S6
```

```
L7:
JOIN cont2, L8
QUIT

L8:
S7
```

FORK – WAIT

```
main ()
{
int pid_hijo;
pid_hijo = FORK();
if (pid_hijo < 0)
        SIGNAL_ERROR
else
{
if (pid_hijo == 0)
      // Soy el hijo
      else
            // Soy el padre
}
}
```

6.3 Hardware de sincronización

Las características de hardware pueden facilitar las tareas de programación y mejorar la eficiencia del sistema.

El problema de las sección crítica se resolverá fácilmente en un entorno uniprocesador, prohibiendo las interrupciones mientras se modifica una variable, pero esa solución no es factible en un entorno multiprocesador, ya que es costoso sincronizar todos los CPU con la prohibición de interrupciones.

Muchas máquinas cuentan con instrucciones de hardware para modificar variables en forma atómica.

La instrucción *Evaluar_y_asignar* (*test_and_set)* se ejecuta atómicamente como unidad ininterrumpible, por lo tanto, si se ejecutan dos a la vez, lo van a hacer secuencialmente y no concurrentemente.

```
function Evaluar_y_asignar (Var objetivo : boolean) : boolean
       begin
             Evaluar_y_asignar:= objetivo;
             objetivo:= true;
       end
```

La forma de usarlo es:

```
while Evaluar_y_asignar (cerradura) do nada;
       SECCIÓN CRÍTICA
cerradura:= false;
       SECCIÓN RESTANTE
```

La instrucción *intercambiar* opera sobre dos palabras, ejecutándose atómicamente.

```
procedure Intercambiar (var a, b: boolean);
       var temp: boolean;
       begin
```

```
            temp := a;
            a:= b;
            b:= temp;
      end;
```

y se usa de la forma:

```
repeat

      llave:= true;
      repeat
            Intercambiar (cerradura, llave);
      until llave = false;
      SECCIÓN CRÍTICA
      cerradura:= false;
      SECCIÓN RESTANTE
until false;
```

6.4 Semáforos

Las soluciones dadas anteriormente no son generalizables, para superar esa dificultad, podemos usar una herramienta de sincronización llamada *semáforo.*
Un semáforo S, es una variable entera, que luego de asignarle un valor inicial, solo se lo puede acceder con dos operaciones atómicas P (espera) y V (señal).

P(S):
while (s ≤ 0);
s--;

V(S):
s++;

Semáforo binario: Su valor varía entre 0 y 1.

6.4.1 Uso

Se puede usar semáforos para resolver problemas de sección crítica para n procesos. Todos los procesos comparten un semáforo de valor inicial 1. Además podemos sincronizar procesos, iniciando S con valor 0 y poniendo:

```
P1;
V(S);
```

Por lo tanto P2, va a ejecutarse después de P1, porque tiene que esperar V(S).

```
P(S);
P2;
```

6.4.2 Implementación

Las desventajas del ejemplo anterior, es que requiere *busy waiting*, ya que cualquier proceso que quiera ingresar a la sección crítica, mientras otro está en ella, debe ejecutar continuamente el ciclo de ingreso, generando un gasto innecesario del CPU, aunque en sistemas multiprocesador puede ser una ventaja, porque no requiere cambio de contexto.

Para no usar busy waiting, se pueden definir los semáforos de otra forma, bloqueando el proceso mientras espera. La operación coloca al proceso en una cola de espera asociada al semáforo, y este proceso se inicia cuando otro proceso ejecute V de ese semáforo. El semáforo se define como un arreglo:

```
type semáforo = record
valor: integer;
L: list of proceso;
end;
```

Y las operaciones son de la forma:

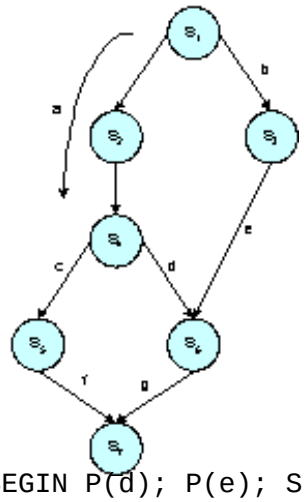
P(S): **if** (s > 0) **then**
s--;
else
espero en S;
endif

V(S): **if** (∃ tarea en espera por S) **then**
despierto una;
else
s++;
endif

P(S) = s--;
if (s < 0) **wait**;

V(S) = s++;
if (s ≤ 0) despierto una;

Ejemplo:




```
VAR a, b, c, d, e, f, g SEMAFORO;
INIT (a, 0);
INIT (b, 0);
INIT (c, 0);
INIT (d, 0);
INIT (e, 0);
INIT (f, 0);
INIT (g, 0);
COBEGIN

BEGIN S1; V(a); V(b); END
BEGIN P(a); S2; S4; V(c); V(d); END
BEGIN P(b); S3; V(e); END
BEGIN P(c); S5; V(f); END
BEGIN P(d); P(e); S6; V(g); END
BEGIN P(f); P(g); S7; END

COEND
```

Otra forma, es tratando de ahorrar semáforos.

```
VAR d SEMAFORO;
INIT (d, 0);

S1;

COBEGIN

BEGIN S2; S4; V(d); S5; END
BEGIN S3; V(d); END
BEGIN P(d); P(d); S6; END

COEND

S7;
```

Se debe crear mutua exclusión entre P y V, para ello, se usa busy waiting, ya que los códigos de P y V son cortos, y la espera es muy pequeña.

La lista de procesos en espera (cuando hago P), se puede implementar de diferentes formas.

6.4.3 Bloqueos mutuos e inanición

Se puede llegar a dar el caso en que dos o más procesos esperen indefinidamente la ocurrencia de un suceso. Estos procesos están en bloqueo mutuo.

Ejemplo:

P_1	**P_2**
Espera(S)	Espera(Q)
Espera(Q)	Espera(S)
.	.
.	.
.	.
Señal(Q)	Señal(S)
Señal(S)	Señal(Q)

P_1 ejecuta Espera(S), y P_2 ejecuta Espera(Q), P_2 no va a poder continuar hasta que P_1 no ejecute Señal(Q), y de la misma forma, P_1 no puede continuar hasta que P_2 ejecute Señal(S). Los dos procesos están en bloqueo mutuo.

Otro problema es la *inanición* que es cuando algunos procesos esperan indefinidamente dentro del semáforo. Puede suceder si la lista de espera del semáforo es LIFO.

6.4.4 Semáforos binarios

En el caso anterior, el semáforo es llamado *de conteo* porque su valor puede tomar cualquier valor entero. Los semáforos *binarios* solo pueden tomar los valores 0 y 1.
Es posible implementar un semáforo de conteo, utilizando semáforos binarios. Se haría de la forma:

Estructura (S):

```
var     S1: SemBin;
        S2: SemBin;
        C: integer;
```

Espera (S):

```
Espera (S1);
C--;
if (C < 0)
{
        Señal (S1);
        Espera (S2);
}
Señal (S1);
```

Señal(S):

```
Espera (S1);
C++;
if (C ≤ 0)
        Señal (S2);
else
        Señal (S1);
```

6.5 Problemas clásicos de sincronización

6.5.1 El problema del buffer

Contamos con un proceso productor que almacena en un buffer, y con un proceso consumidor, que consume del buffer.

Buffer infinito

```
PROCEDURE Prod                    |  PROCEDURE Cons
REPEAT                            |  REPEAT
        P(s);                     |          P(n);
        Agregar(Buffer.dato);     |          P(s);
        V(s);                     |          Sacar (Buffer.dato);
        V(n);                     |          V(s);
FOREVER                           |  FOREVER
```

```
BEGIN
        INIT (n, 0);
        INIT (s, 1);
        COBEGIN
                Prod; Prod; ...
                Cons; Cons; ...
        COEND
END
```

Buffer finito:

```
PROCEDURE Prod                    |  PROCEDURE Cons
REPEAT                            |  REPEAT
        P(l);                     |          P(n);
        P(s);                     |          P(s);
        Agregar(Buffer.dato);     |          Sacar (Buffer.dato);
        V(s);                     |          V(s);
        V(n);                     |          V(l);
FOREVER                           |  FOREVER
```

```
BEGIN
        INIT (n, 0);
        INIT (s, 1);
        INIT (l, tam_buff);
        COBEGIN
                Prod; Prod; ...
                Cons; Cons; ...
        COEND
END
```

6.5.2 El problema de los lectores y escritores

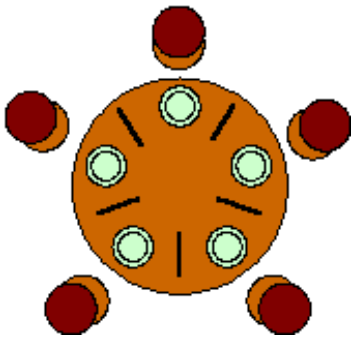
Un objeto de datos se comparte entre procesos, los cuales pueden leer o escribir. Si dos lectores acceden simultáneamente no pasa nada malo, sin embargo si un escritor y otro proceso (lector o escritor) acceden simultáneamente, el resultado puede ser erróneo.
Hay dos planteos. El primero es el de no tener esperando a ningún lector a menos que el escritor ya esté escribiendo. El segundo quiere que apenas haya un escritor, realice su escritura lo antes posible.

```
PROCEDURE Lector
REPEAT
        P(mutex);
        ContLeer++;
        If ContLeer = 1 then P(esc);
        V(mutex);

        LEER

        ContLeer--;
        if ContLeer = 0 then V(esc)
        V(mutex);
FOREVER
```

```
PROCEDURE Escritor
REPEAT
        P(esc);

        ESCRIBIR

        V(esc);
FOREVER
```

```
BEGIN
        INIT (esc, 1);
        INIT (mutex, 1);
        INIT (I, tam_buff);
        COBEGIN
                Lect; Lect; ...
                Esc; Esc; ...
        COEND
END
```

6.5.3 El problema de los filósofos



Hay cinco filósofos que se pasan la vida pensando y comiendo. Comparten una mesa circular rodeada por cinco sillas, con un plato con arroz para cada uno. Además hay cinco palillos chinos individuales.
Cuando el filósofo piensa, no interactúa, pero cuando siente hambre, trata de tomar dos palillos para comer, y obviamente no puede tomar un palillo que esté en manos de sus colegas. Cuando termina de comer deja ambos palillos sobre la mesa y comienza a pensar otra vez.

Una forma es representar cada palito con un semáforo.

Esta posibilidad no permite que dos vecinos coman simultáneamente, pero puede generar bloqueos mutuos, por ejemplo si todos sienten hambre a la vez y toman su palillo izquierdo. Algunas formas de solucionarlo son:
Permitiendo que solo cuatro filósofos se sienten a la vez
Solo permitir que tome los palillos, cuando los dos están disponibles (tomándolos dentro de una sección crítica)
Solución asimétrica, donde los filósofos pares toman primero el de la derecha, y los impares primero el de la izquierda.

```
PROCEDURE Filósofo (I : Integer)
        izq = I;
        der = (I + 1) mod 5;
        REPEAT
                PENSAR;
                P(Tenedor(izq));
                P(Tenedor(der));
                COMER;
                V(Tenedor(der));
                V(Tenedor(izq));
        FOREVER
END Filósofo


BEGIN
        Tenedor::array(0..4) of SEMAFORO;
        for (int I = 1, I < 4; i++)
                init (Tenedor(i), 1);
        COBEGIN
                for (int j = 1, j < 4; i++)
                Filósofo(j);
        COEND
END
```

Solo cuatro filósofos a la vez:

```
PROCEDURE Filósofo (I : Integer)
        izq = I;
        der = (I + 1) mod 5;
        REPEAT
                PENSAR;
                P(Comedor);
                P(Tenedor(izq));
                P(Tenedor(der));
                COMER;
                V(Tenedor(der));
                V(Tenedor(izq));
                V(Comedor);
        FOREVER
END Filósofo

BEGIN
        Tenedor::array(0..4) of SEMAFORO;
        for (int I = 1, I < 4; i++)
                init (Tenedor(i), 1);
        init (Comedor, 4);
        COBEGIN
                for (int j = 1, j < 4; i++)
                Filósofo(j);
        COEND
END
```

6.6 Regiones críticas

Los semáforos pueden dar pie a errores de temporización que ocurren cuando se dan ciertas secuencias de ejecución específica. Para reducir la posibilidad de ocurrencia de estos errores se han introducido varias construcciones en los lenguajes de alto nivel, la *región crítica*.

Suponemos que un proceso consiste en algunos datos locales y un programa secuencial que puede operar sobre esos datos. Solo el programa secuencial de un proceso puede acceder a sus datos, o sea, que no puede acceder directamente a los datos locales de otro proceso. Los procesos si pueden compartir datos globales.
La forma de hacer esto es: Si tengo una variable **v**, que puedo cambiar, debo tener otra variable **B** global. Solo puede accederse a v, cumpliendo el enunciado ***region*** *v* ***when*** *B* ***do*** *S*. Cuando el proceso intenta acceder a v, debe cumplir B para lograr ingresar a la sección crítica. Esta solución de todas formas, no contempla algunos problemas de sincronización, aunque reduce su número.

6.7 Monitores

Un monitor se caracteriza por un conjunto de operadores definidos por el programador. Consiste en declaración de variables cuyos valores definen el estado de un ejemplar del tipo, así como los cuerpos de procedimientos o funciones que implementan operaciones con el tipo.

La sintaxis es:

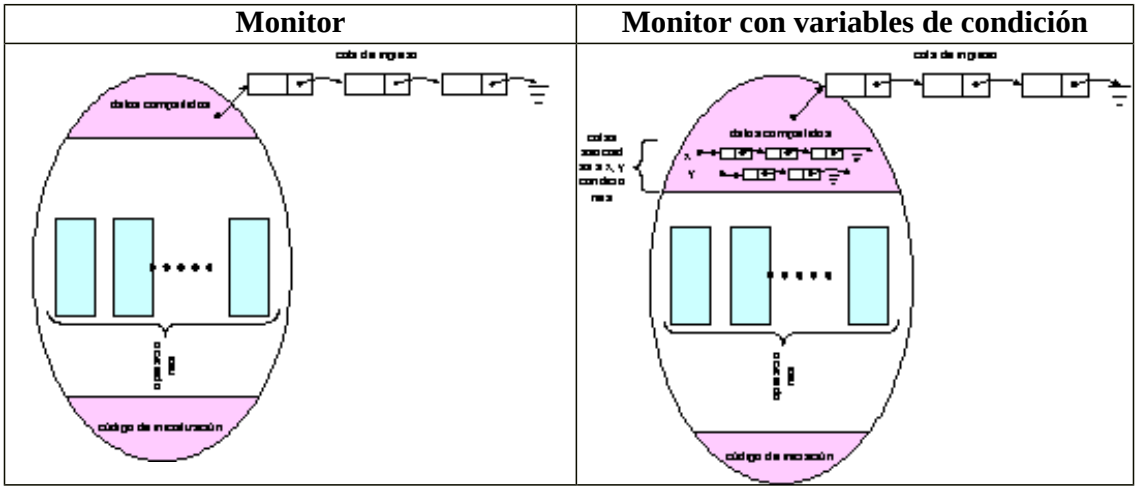
```
type nombre_monitor = monitor
       declaración de variables

procedure entry P1 (...)
       begin … end;

procedure entry P2 (...)
       begin … end;

……

procedure entry Pn (...)
       begin … end;

begin
       código de inicialización
end.
```

Un procedimiento definido dentro de un monitor solo puede acceder a las variables definidas dentro del monitor, por lo tanto el programador no necesita codificar la restricción de sincronización.
La construcción del monitor garantiza que solo un proceso a la vez puede estar activo; sin embargo esta construcción todavía no logra modelar algunos esquemas de sincronización.

La construcción *condición* proporciona dichos mecanismos

var x, y : condición;

y las únicas operaciones que puede invocar x, son x.espera y x.señal, donde x.señal, reanuda uno y solo un proceso suspendido, y si no hay ninguno, no tiene efecto.



Si P y Q están asociados a x, y P ejecuta x.señal, Q renuda su ejecución, por lo tanto P debe esperar.
Cuando P ejecuta x.señal, inmediatamente sale del monitor, por lo tanto Q también reanuda de inmediato.

Filósofos comensales:

var estado : **array**[0..4] **of** (pensando, hambriento, comiendo);

donde un filósofo puede estar comiendo, solo si no lo hacen sus vecinos. También se necesita declarar:

var mismo : **array**[0..4] **of** condición;

por la cual puedo retardar al filósofo *i* si tiene hambre, pero no puede obtener los palillos.

```
type filósofos_comensales = monitor
       var estado : array [0..4] of (pensando, hambriento, comiendo);
       var mismo : array [0..4] of condición;

       procedure entry tomar (i : 0..4);
             begin
                    estado[i] := hambriento;
                    probar(i);
                    if estado[i] <> comiendo then mismo[i].espera;
             end;

       procedure entry dejar (i : 0..4);
             begin
                    estado[i] := pensando;
                    probar(i + 4 mod 5);
                    probar(i + 1 mod 5);
             end;

       procedure entry probar (k : 0..4);
             begin
                    if estado[k+4 mod 5] <> comiendo;
                           and estado[k] = hambriento;
                           and estado[k+1 mod 5] <> comiendo;
                    then   begin
                                  estado[k] := comiendo;
                                 mismo[k].señal;
                           end;
                    end;
       begin
             for i:= 0 to 4
                    do estado[i] := pensando;
       end.
```

Cada filósofo antes de comenzar a comer, debe invocar la operación tomar. El proceso se puede suspender, y una vez que la operación tiene éxito, el filósofo puede comer. A continuación el filósofo invoca *dejar* y puede comenzar a pensar.
Esto soluciona que dos vecinos no estén comiendo simultáneamente, y que no ocurren bloqueos mutuos, pero un filósofo podría morir de hambre.
Otro problema es el orden de reanudación de procesos suspendidos por x, si alguno ejecuta x.señal. Una forma de hacerlo es con FCFS.

Productor consumidor:

```
const tamaño 184
Monitor buff_finito
var
       b: array [0..tamaño] of integer;
       n, in, out: integer;
       nolleno, novacio: condition;

procedure Agrego (v: integer)
begin
       if (n == tamaño + 1) then
             nolleno.wait()
       endif

       b[in] = v;
       in = (in + 1) mod (tamaño + 1);
       n++;
       novacio.signal();
end

procedure Saco (v: integer)
begin
       if (n == 0) then
             novacio.wait()
       endif

       v = b[out];
       out = (out + 1) mod (tamaño + 1);
       n--;
       novacio.signal();
end

begin
      n = 0; in = 0; out = 0;
end
```

Lector Escritor:

```
program Lector_Escritor

Monitor LE;
var
       lect : integer;
       esc : boolean;
       okleer, okescribir : condition;

procedure com_lect

       if (esc or NonEmpty (okescribir)) then
             okleer.wait;
             endif
       lect++;
       okleer.signal;
end

procedure fin_lect

       lect--;
```

```
        if (lect = 0) then
            okescribir.signal;
        endif
end

procedure com_esc

        if (lect <> 0 or esc) then
            okescribir.wait;
        endif

        esc = true;
end

procedure fin_esc

        esc = false;
        if (NonEmpty (okescribir)) then
            okescribir.signal;
        else
            okleer.signal;
        endif
end

begin
        lect = 0; esc = false;
end

procedure Lectura

        com_leer
        LEER
        fin_leer
end

procedure Escribir

        com_esc
        ESCRIBIR
        fin_esc
end

begin
        cobegin
            Escribir; Escribir; …
            Leer; Leer; …
        coend
end
```

6.8 Sincronización en Solaris 2

Antes de Solaris 2, el sistema implementaba las secciones críticas estableciendo el nivel de interrupción, a un valor tanto o más alto que cualquier interrupción que pudiera modificar los mismos datos.
Solaris 2, se diseñó con miras a ofrecer capacidades de tiempo real, ser miltihilado y manejar multiprocesadores. Si hubiera seguido utilizando secciones críticas, el desempeño hubiera sufrido una degradación considerable, pues el núcleo se habría convertido en un cuello de botella.

Solaris 2 emplea *mutex adaptativos*, que inicialmente es un semáforo estándar implementado como cerradura de giro.

6.9 Transacciones atómicas

La mutua exclusión de las secciones críticas asegura que estas se ejecutarán atómicamente.

6.9.1 Modelo del sistema

Una colección de instrucciones que realiza una sola función lógica, es una transacción. Un aspecto importante es la preservación de atomicidad. Una transacción es una unidad de programa que accede y actualiza datos. Es una secuencia de operaciones “leer” y “escribir” que termina con una operación “confirmar” si terminó con éxito, o “abortar”, si hubo error o un fallo del sistema.

Una transacción abortada podría haber modificado datos, por lo tanto, esos datos deben ser restaurados; se “revierte” la operación. Una de las obligaciones del sistema es garantizar esa propiedad.

La forma que utiliza el sistema para garantizar la atomicidad depende del medio de almacenamiento.

- **Almacenamiento volátil:** La información casi nunca sobrevive a una caída del sistema. Un ejemplo es la memoria principal o el caché.
- **Almacenamiento no volátil:** La información por lo regular sobrevive a la caída del sistema. Un ejemplo son los discos o cintas magnéticas.
- **Almacenamiento estable:** La información nunca se pierde.

6.9.2 Recuperación basada en bitácoras

El sistema mantiene en el almacenamiento estable una estructura de datos llamada *bitácora*, donde cada registro, describe una sola operación de escritura, y tiene los siguientes campos:

- Nombre de la transacción
- Nombre del dato
- Valor antiguo del dato
- Valor nuevo del dato

y también almacena <T_i inicia> y <T_i confirma> que controla el inicio de la escritura y el fin exitoso.

Nunca podemos permitir que se actualice un dato en una transacción, antes de registrarlo en la bitácora, por lo tanto se requieren dos escrituras físicas por cada escritura lógica.

El algoritmo de recuperación emplea dos procedimientos: deshacer (T_i) y rehacer (T_i), las cuales deben ser idempotentes para garantizar un comportamiento correcto.

- La transacción T_i se deshace, si la bitácora contiene <T_i inicia> y no contiene <T_i confirma>.
- Si contiene las dos, la transacción debe rehacerse.

6.9.3 Puntos de verificación (checkpoints)

Cuando ocurre un fallo, se necesita consultar la bitácora, para ver que transacciones rehacer, y cuales deshacer, pero en un principio hay que examinarla toda. Esto tiene dos desventajas:

1. El proceso de búsqueda consume mucho tiempo.

2. La mayor parte de las transacciones que deben rehacerse ya actualizaron los datos que hay que modificar.

Para reducir gastos extra se introduce el concepto de checkpoint, que realiza la secuencia de acciones siguiente:

1. Grabar en almacenamiento estable los registros de la bitácora que están en almacenamiento volátil.
2. Grabar en el almacenamiento estable todos los datos modificados que residen en almacenamiento volátil.
3. Grabar un registro de bitácora <checkpoint> en almacenamiento estable.

Cuando ocurra un fallo, vamos a rehacer y deshacer las transacciones que fueron hechas después del último checkpoint.

6.9.4 Transacciones atómicas concurrentes

Las transacciones deben ser atómicas, y esto se realiza ejecutando cada transacción dentro de una sección crítica, usando semáforos. Esto soluciona el problema pero es demasiado restrictivo, puesto que hay transacciones que si se pueden ejecutar concurrentemente.

6.9.4.1 Seriabilidad

T_0	T_1
leer(A)	
escribir (A)	
leer (B)	
escribir (B)	
	leer (A)
	escribir (A)
	leer (B)
	escribir (B)

Consideramos un sistema con dos datos A y B, y las transacciones T_0 y T_1 que leen y escriben. Si las ejecutamos atómicamente sería à

Este plan de ejecución atómica se lo denomina “plan serial”. En este caso la operación escribir(A) de T_0 está en conflicto con la operación leer(A) de T_1, en cambio, no lo está con la leer(B) de T_1. Cuando dos operaciones no están en conflicto se las puede intercambiar, creando un nuevo plan. Obtenemos:

T_0	T_1
leer(A)	
escribir (A)	
	leer (A)
	escribir (A)
leer (B)	
escribir (B)	
	leer (B)
	escribir (B)

6.9.4.2 Protocolo de cerraduras

Una forma de asegurar la seriabilidad, es asociar a cada dato una cerradura, y cada transacción se debe regir por un protocolo de cerraduras. Hay dos modos:

- **Compartido:** T_i Puede leer el dato pero no escribirlo.
- **Exclusivo:** T_i puede leer y escribir el dato.

En este caso, no aseguramos la seriabilidad. Un protocolo que si lo hace es el *protocolo de cerraduras de dos fases*, pero en este caso no aseguro la ausencia de deadlock. Las fases son:

- **Fase de crecimiento:** Una transacción puede obtener cerraduras pero no liberar.
- **Fase de encogimiento:** Una transacción puede liberar cerraduras, pero no obtener.

6.9.4.3 Protocolos basados en marcas de tiempo

En los protocolos anteriores, el orden de cada par de transacciones en conflicto, se determina en tiempo de ejecución. Otro método para mantener la seriabilidad consiste en escoger con anticipación un ordenamiento.
El ordenamiento por marcas de tiempo, asigna a cada transacción una marca de tiempo antes de que se ejecute, y si más adelante entra otra transacción, a la última se le asigna una marca de tiempo mayor. Para hacerlo hay dos métodos:

- Utilizar el reloj del sistema como marca de tiempo. Este método no funciona en procesadores que no comparten el mismo reloj.
- Utilizar un contador lógico.

A cada dato Q se le asignan dos valores de marca de tiempo:

- MT-E(Q): Marca la última escritura de Q.
- MT-L(Q): Marca la última lectura de Q.

El protocolo es:

- T_i solicita **leer(Q)**:
  - Si TS(T_i) (marca de tiempo asignada a T_i) < MT-E(Q): T_i necesita leer un valor de Q que ya se sobrescribió, por lo tanto la lectura se rechaza y T_i se revierte.
  - Si TS(T_i) ≥ MT-E(Q): Se ejecuta leer(Q) y se asigna MT-L(Q) = max {MT-L(Q), TS(T_i)}
- T_i solicita **escribir(Q)**:
  - Si TS(T_i) < MT-L(Q): El valor de Q que T_i está produciendo se necesitó previamente, por lo tanto escribir(Q) se rechaza y T_i se revierte.
  - Si TS(T_i) < MT-E(Q): T_i está intentando escribir un valor obsoleto de Q, por lo tanto escribir(Q) se rechaza y T_i se revierte.
  - En caso contrario, escribir(Q) se ejecuta.

Ada

Originalmente diseñado para aplicaciones militares, Ada es un lenguaje de propósito general que puede ser usado para cualquier problema. Tiene una estructura de bloque y un mecanismo de tipo de datos igual que Pascal, aunque con extensiones para aplicaciones de tiempo real y distribuidas.

Un poco de historia

En los 70's hubo interés del Departamento de Defensa de E.U.A. para desarrollar un lenguaje sencillo para usar en sistemas de tiempo real incrustados.
Pascal fue el punto de partida para el diseño de ADA pero el lenguaje resultante es muy diferente en muchos aspectos. Ada es más extenso, más complejo, permite ejecución concurrente, control en tiempo real de la ejecución, manejo de excepciones y tipos de datos abstractos.

Pequeña descripción del lenguaje

Ada esta hecho para soportar la construcción de grandes programas. Un programa en Ada esta ordinariamente diseñado como una colección de grandes componentes de software llamados "Packages" cada uno representando un tipo de dato abstracto o un conjunto de objetos de datos compartidos entre subprogramas. Un programa en Ada consiste de un procedimiento singular que sirve como programa principal, el cual declara variables, y ejecuta sentencias, incluyendo llamadas a otros subprogramas. Un programa Ada puede envolver tareas que se ejecuten concurrentemente, si esto pasa entonces estas son inicializadas directamente por el programa principal y forman el nivel superior de la estructura del programa.

Ada provee un gran número de tipos de datos, incluyendo enteros, reales, enumeraciones, booleanos, arreglos, records, cadena de caracteres y apuntadores. Abstracción y encapsulación de tipos de datos y operaciones definidas por el usuario son provistas por la característica de "package". El control de secuencia dentro de un subprograma utiliza expresiones y estructuras de control similares a Pascal. La estructura de control de datos de Ada utiliza la organización de estructura de bloque estática como en Pascal y además el lenguaje prosee llamadas a referencias no estáticas. Desde que los programas pueden ejecutar tareas concurrentemente, estos pueden correr subprogramas independientemente uno del otro.

Tipos de datos primitivos

Variables y Constantes

- Cualquier objeto de datos puede ser definido como variable o como constante.
- Cualquier declaración que empiece con la palabra **constant** es una constante y se deberá asignar un valor el cual no se podrá cambiar en la ejecución.
- Si se omite la palabra **constant** la misma declaración define un tipo de dato variable, a la cual se le debe asignar un valor inicial y el cual se podrá cambiar en la ejecución.

Ejemplo:
MaxSize **constant** integer := 500;
CurrentSize integer := 0;

Tipos de Datos Numéricos

- Enteros, punto-flotante, punto-fijo y los tipos de datos básicos.
- Declaraciones son similares que en Pascal usando los atributos **range** (para enteros) y **digits** (para flotantes) los cuales especifican el rango de valores que podra tomar el tipo de dato.

Ejemplo:

```
type DayOfYear is range 1..366; -- un valor entero de 1 a 366
MyBirhtday: DayOfYear := 219;  -- MyBirthday inicializada a 219
type Result is digits 7                    -- flotante de 7 digitos
Anwer: Result :=3.17;                              -- varible de 7 digitos.
```

Los tipos *Integer* y *float* están predefinidos en el paquete estándar como:

type integer **is range** *implementation defined*;
type float **is digits** *implementation defined*;

Enumeraciones

- Enumeraciones deben de definirse usando un estilo como Pascal de definición e implementación.

Ejemplo:

```
type class is (Fresh, Soph, Junior, Senior);
```

La representación en la ejecución usa el número de posición para cada valor de la literal, empezando con 0 para el primer valor listado, 1 para el segundo etc.

Tipos Caracter y Boleano

- Estas son definidas en el paquete estándar para ser enumeraciones específicas.

Tipo de Dato Apuntador

- Un tipo apuntador llamado **access**, junto con una función primitiva **new**, crean un nuevo objeto y regresa un apuntador a este, el cual debe ser asignado a una variable del tipo **access**.
- Una variable no pude ser directamente declarada como **access** en vez de esto se hace lo siguiente:

type acces_typename **is access** typename;

y la variable deber ser declarada como un tipo de dato definido.

- Todas las variables del tipo **acces** tienen el valor NULL por default.
- Un bloque de almacenamiento puede ser solicitado para cualquier tipo de dato usando la sentencia:

for *acces_typename* **use** *expresion*;

donde expresión es el tamaño del bloque a pedir.

Tipo de datos estructurados

Vectores y Arreglos

- Un arreglo de objetos puede ser declarado con cualquier número de dimensiones, cualquier rango y cualquier tipo de componente.

Ejemplo:

Table: **array** (1..10, 1..20) of float;

crea una matriz de 10 X 20 de números reales;

- Definición de tipos puede ser utilizada para crear clases de cualquier tipo de arreglo de objetos.

Ejemplo:

type Matrix **is array** (integer **range** <>,integer **range** <>) of float;

donde los < > indican un campo que se debe llenar.
Inicialización: cualquier arreglo deber de ser inicializado en su declaración.

Cadena de Caracteres

- Las cadenas de caracteres son tratadas como un tipo de vector predefinido usando otros dos tipos predefinidos positive (enteros) y character (enumerando los caracteres definidos en el paquete estandar).

type string **array** (positive **range** <>) of character;

Ejemplo:

MyString : string(1..30);

Tipo de Dato Archivo

- Archivos y operaciones de entrada-salida son definidos como tipos de datos abstractos, usando paquetes predefinidos.
- El programador de Ada mira los archivos como tipo de datos que en su estructura interna están ocultos por ser encapsulados con los paquetes estándares.
- Existen diferentes tipos de archivos (texto,enteros,reales,etc..) cada paquete tiene su propio tipo de archivo ejemplo en TEXT_IO hay un tipo para archivos secuenciales o solo texto en FLOAT_IO hay un tipo de dato archivo para flotantes etc.

Tipo de Dato Definido Por El Usuario

- Los tipos de datos record en Ada son similares que en Pascal.
- En Ada a diferencia de Pascal un objeto record debe de ser definido primero usando una definición de tipo, para definir la estructura de record y después dando la declaración de la variable usando el tipo de nombre.

Ejemplo:

```
type Birthday is
   record
            Month: string (1..3) := "jun";
                     Day:  integer range 1..31 := 17;
Year: integer range 1950..2050 := 1960;
   end record
```

cada variable del tipo Birthday esta inicializada a "jun" 17 1960.

Control de secuencia

La ejecución de Ada esta orientada a sentencias como FORTRAN y Pascal.

Expresiones

- Expresiones en Ada permiten primitivas y funciones definidas para ser combinadas en secuencias de manera usual.
- Notación infija es usada para la aritmética binaria, relacional y operaciones boleanas.
- Notación prefija es usada para operaciones unarias. (+,-, y NOT), los paréntesis deben de utilizarse para agrupar operaciones y la precedencia es la misma que en Pascal.

Sentencias

- Las estructuras de secuencias de control de sentencias son las usuales sentencias condicionales (if y case) y las sentencias de iteración (loop).
- Todas las sentencias de control terminan con las sentencia end seguida con la palabra de dicha sentencia ejem: end if, end loop.

Asignacion: La asignación es similar que en Pascal. La asignación completa de un grupo de valores a una variable record puede ser posible.

Ejemplo:

```
MyDay := ("SEP",12,1975);
MyDay := (Year => 1975, Month =>"SEP", Day => 12);
```

Sentencias condicionales

Sentencia If

- La sentencia if tiene la siguiente forma:

```
if boolena expresion then
        - secuencias de sentencias
elsif boolean expresion then
        - secuencia de sentencias
elsif boolean expresion then
        - secuencia de sentencias
...
else
        - secuencia de sentencias
end if
```

Sentencia CASE

- La forma general de case es:

```
case expresion is
 when choice | ...| choice => secuencia de sentencias;
 when choice | ...| choice => secuencia de sentencias;
...
 when others =>  secuencia de sentencias;

end case;
```

Cada choice indica o uno o varios rangos posibles de valores para la expresión al principio de la sentencia.

Ejemplo:

```
case GradeLevel is
 when Fresh => sentencias;
 when Soph | Junior => secuencias;
 when Senior => secuencias;
end case;
```

Sentencias de iteración

Sentencia Loop

- La sentencia básica de iteración tiene la siguiente forma.

```
loop
  - secuencia de sentencias
end loop
```

- La forma de terminar una sentencia de iteración **loop** se hace explícitamente con las sentencias **exit**, **goto** o **return**.
- Sentencias controladas pueden ser creadas con las sentencias **while** y **for**.

while *boolean expresion*
...

con **for** hay dos formas.

for variable name **in** discrete_range;
for variable name **in reverse** discrete_range;

Subprogramas y manejo de almacenamiento

Procedimientos, funciones y tareas son los tres tipos de subprogramas que Ada maneja.

Funciones y Procedimientos

- Un procedimiento o función tiene la forma de una especificación:

procedure procname(*formal-parameters*) **is** ...
function functioname(*formal-parameters*) return *result_type* **is**

seguida por un cuerpo como sigue :

```
  - secuencia de declaraciones
begin
  -secuencia de sentencias
exception
  - manejadores de excepciones
end
```

- Los subprogramas pueden ser recursivos. La salida normal de un procedimiento o tarea es con la ejecución de **return**;
- En el enunciado formal-parameters deben de ir los tipos de datos con su nombre y su forma de transmisión para ser usado.
  - Excepciones: Al final de cada unidad de programa debe de especificarse un manejador de excepciones. Existen varias excepciones predefinidas en los paquetes estándares, todas las demás son declaradas usando la siguiente

sentencia:

exceptionname: **exception**

- Un manejador de excepciones empieza con el nombre de la excepción que este maneja, seguida con una secuencia de sentencias que toman la acción apropiada para manejar dicha excepción.

exception
when *exception_name* | ... | *exception_name*
=> secuencia de sentenecias
...
when *others* => secuencia de sentencias

donde cada secuencia de sentencias deberá manejar una o mas excepciones nombradas.

Tareas: Una tarea es un subprograma que puede ejecutarse concurrentemente con otras tareas.

- La definición de una tarea tiene una forma general, tiene una especificación que permite a otras tareas comunicarse entre si y un cuerpo de implementación. Usando un tipo de tarea, múltiples tareas pueden ser creadas con una definición singular.

task *taskname* **is**
-declaraciones de entrada
end
task body *taskname* **is**
-secuencia de declaraciones
begin
-secuencia de setencias
exception
-manejador de excepciones
end;

Abstracción y encapsulamiento

Paquetes

- La definición de un paquete tiene dos partes una especificación y un cuerpo.La especificación contiene la información necesaria para la correcta utilización del paquete; el cuerpo proveee la encapsulación de variables y subprogramas que son llamados desde afuera.
- La parte visible de un paquete (todo lo que procede a la palabra **private**) define que es lo que el usuario final de el paquete puede usar en un programa.

package *packagename* **is**
-declaración de variables
private
-definición completa de datos privados
end;
packages body packagename **is**
-definiciones de objetos y subprogramas definidos arriba

begin
-sentencias para inicializar el paquete cuando es por primera

vez instanciado

exception
- manejador de excepciones
end;

Hola Mundo!!

```
% vi Hola.ada
  with Text_IO; use Text_IO;
procedure Doit is
   procedure Hola(x: in integer) is
   begin
   if x=2 then
       put("Hola Mundito");
  else  put ("Adios Mundito");
  end if
  end Hola;
y: integer;
begin
y:=2;
Hola(y);
end Doit;
% ada Hola.ada
%Hola
Hola Mundito
```

Ejemplos de procedimientos:

```
Procedure Algo ()
// Los task son como procedimientos
TASK …
…
end
Begin
…
end Algo
TASK Uno is
     entry ident + 1 ()  à  Pueden tener o no parámetros
end Uno  à  Puede tener más citas (entry's)

TASK body Uno is
// Aceptar citas
Accept ident1 ()
                  sentencias
end ident1
end Uno

Accept ident1 (A: IN integer; B: OUT String; C: INOUT integer)
…
end Accept

// Invocación
```

Uno Ident1 (2, P, R)

Alicia – Bernardo:

```
TASK Patio
  ENTRY Pedir
  ENTRY Devolver
END

TASK BODY Patio
BEGIN
  loop
           Accept Pedir;
           Accept Devolver;
  end loop
END Patio
```

Factorial:

```
PROCEDURE Factorial IS

PROCEDURE Print (I: IN integer) is separate; à Compilado en forma separada
PROCEDURE Get (I: OUT integer) is separate;
n, x, y : integer;

TASK Type Semifact is
  entry Num (I: IN integer)
  entry Resultado (I: OUT integer)
END Semifact
TASK BODY Semifact is separate;
A, B : Semifact;
BEGIN
  Get (N) à Obtengo el número del teclado
  A.Num (N)
  B.Num (N-1)
  A.Resultado (x)
  B.Resultado (y)
  Print (x * y)
END
TASK BODY Semifact is
  u: integer;
BEGIN
  Accept Num (I: IN integer)
           u:= I
  end Num
  Semifactorial (u) à Hay que especificar este procedimiento aparte
  Accept Resultado (I: OUT integer)
           I = u;
  end Resultado
end Semifact
```

Productor – Consumidor:

```
TASK Buff_Simple is
  entry Leer (I: OUT character)
  entry Escribir (I: IN character)
end Buff_Simple
```

```
TASK BODY Buff_Simple is
  n: integer
  x: character
BEGIN
  loop
          Accept Escribir (A: IN character) do
                  x:= A;
          end Escribir
          Accept Leer (A: OUT character) do
                  A:= x;
          end Leer
  end loop
end Buff_Simple

TASK Productor;

TASK BODY Productor is
  c1: character
BEGIN
  loop
          Produce (c1)
          Buff_Simple.Escribir(c1)
  end loop
end Productor

TASK Consumidor;

TASK BODY Consumidor is
  c1: character
BEGIN
  loop
          Buff_Simple.leer (c1)
          Consumir(c1)
  end loop
end Consumidor

SELECT à Espero por las dos, sino hay solicitud, queda bloqueado esperando
  Accept Leer (A: OUT character) do
          A:= x
  end Leer
          …
          // Puede haber muchas OR
          …
          OR Accept Escribir (A: IN character) do
                  x:= A
          end Escribir
```

Select:

El select primero evalúa las condiciones, y para las que dio TRUE se queda esperando.

Select
[when cond1 è]
Accept entrada1
…
end Accept
OR

[when cond2 è]
Accept entrada2
…
end Accept
…
…
ELSE (*)
…
end select

(*) También puedo poner OR [when condn = x] delay s, y espera s segundos.

Semáforo:

```
TASK type Semaforo is
  entry Init (I: IN integer);
  entry p;
  entry v;
end semaforo

TASK body semaforo is
  N: integer
begin
  accept Init (I : IN integer) do
          N:= I;
  end accept
  loop
          select
                  when N>0 =>
                          accept p;
                          N = N – 1;
          or
                  accept v;
                  N = N + 1;
          end select
  end loop
end semaforo
```

Productor – consumidor:

```
TASK Buff_finito is
  entry sacar (d:OUT dato)
  entry agregar (d: IN dato)
end Buff_finito

TASK body buff_finito is
  max : constant := 20
  buff: array (0..max) of dato
  ocupado, in, out: integer range 0..max:=0
begin
  loop
          select
                  when ocupado < max =>
                          accept agregar (d: IN dato) do
                                  buff (in):= d
                          end agregar
```

```
                        ocupado:= ocupado + 1;
                        in:= (in + 1) mod max;
          or       when ocupado > 0 =>
                        accept sacar (d: OUT dato) do
                                d:= buff (out)
                        end sacar
                        ocupado:= ocupado – 1;
                        out:= (out + 1) mod max;
          end select
  end loop
end buff_finito
```

En ada, además de poder declarar arrays de tareas, puedo declarar arrays de entradas.

entry entrada: array (0..100) equivale a:
entry entrada (0)
entry entrada (1)
…
entry entrada (100)

7 Bloqueos mutuos (Deadlock)

Cuando un proceso A solicita un recurso que está ocupado por otro proceso B, pasa a un estado de espera. Si B también está en estado de espera, esperando un recurso que tiene A, los dos procesos están en deadlock.

7.1 Modelo del sistema

Un sistema consiste en un número finito de recursos que deben distribuirse entre varios procesos que compiten. Los recursos se dividen en varios tipos, cada uno de los cuales consiste en cierta cantidad de ejemplares idénticos (unidades de recursos). Si un proceso solicita un ejemplar de un tipo de recurso, la asignación de cualquier ejemplar del tipo, satisface la solicitud.
Un proceso solo puede utilizar un recurso, siguiendo esta secuencia:

1. **Solicitud:** Si la solicitud no puede atenderse de inmediato, el proceso debe esperar.
2. **Uso:** El proceso opera con el recurso.
3. **Liberación:** El proceso libera el recurso.

La solicitud y liberación, son llamadas al SO. Una tabla del sistema registra si cada recurso está libre o asignado. Si un proceso solicita un recurso ocupado, se lo coloca en la cola de procesos que esperan ese recurso.

Los recursos pueden ser físicos (impresoras, memoria, ciclos de CPU), o lógicos (archivos, semáforos, monitores).

7.2 Caracterización de bloqueos mutuos

En un bloque mutuo, los procesos nunca terminan su ejecución y los recursos del SO quedan acaparados.

7.2.1 Condiciones necesarias

Ocurre bloqueo mutuo si se cumplen simultáneamente las cuatro condiciones siguientes:

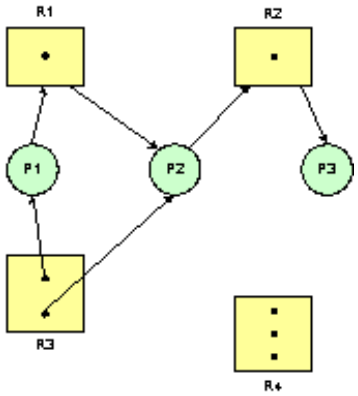
- **Mutua exclusión:** Al menos un recurso debe adquirirse de forma no compartida, o sea, solo puede usarlo un proceso a la vez. Si otro proceso lo solicita, debe esperar.

- **Retener y esperar:** Debe existir un proceso que haya adquirido al menos un recurso y esté esperando para adquirir recursos adicionales, que ya fueron otorgados a otros procesos.
- **No expropiación:** Los recursos no se pueden arrebatar, es decir, la liberación es voluntaria por parte del proceso que adquirió el recurso.
- **Espera circular:** Debe existir un conjunto $\{P_0, P_1, \dots, P_n\}$ de procesos en espera tal que P_0 espera un recurso adquirido por P_1, P_1 espera uno adquirido por P_2, … , P_n espera uno adquirido por P_0.

7.2.2 Grafo de asignación de recursos

Los bloqueos mutuos se pueden describir con mayor precisión en términos del *grafo de asignación de recursos del sistema*. Los vértices son P y R (procesos y recursos). Una arista de P a R indica que el proceso espera el recurso, y una de R a P, indica que se asignó el recurso al proceso.
P se identifica con un círculo, R con un cuadrado, y puntos adentro, donde cada punto es una unidad de recurso.

Casos:



- **No contiene ciclos:** No existe bloqueo mutuo.
- **Si contiene ciclos:**
  - Si los recursos solo tienen un único ejemplar, hay bloqueo mutuo
  - Si hay varios ejemplares, no necesariamente hay bloqueo.

7.3 Métodos para manejar bloqueos mutuos

- Usar un protocolo que asegure que el SO nunca llegará a deadlock
- El sistema entra en un bloqueo mutuo y se recupera
- Desentenderse del problema y hacer como si nunca ocurrieran bloqueos (es lo que hacen la mayor parte de los SO)

Prevención de bloqueos mutuos: Métodos que garantizan que no se cumpla por lo menos una de las condiciones necesarias.

Evitación de bloqueos mutuos: Requiere proporcionar al SO información anticipada acerca de los recursos que el proceso va a solicitar y usar. Con esa información el SO decide si el proceso va a esperar o no.

Cuando el SO no cuenta con ninguno de los dos algoritmos anteriores, pueden ocurrir bloqueos, los cuales deben ser detectados y revertidos, de otra forma, el SO se deteriora hasta dejar de funcionar.

7.4 Prevención de bloqueos mutuos

Podemos tratar de que no se cumpla alguna de las condiciones necesarias para bloqueos mutuos.

7.4.1 Mutua exclusión

Se debe cumplir mutua exclusión para recursos no compartibles. Por ejemplo, dos o más procesos no pueden compartir una impresora, pero si pueden compartir un archivo de solo lectura.

7.4.2 Retener y esperar

Siempre que un proceso solicite un recurso, no debe retener otro. Una forma es que el proceso solicite y reciba todos los recursos antes de iniciar su ejecución. Otra forma es que pueda solicitar recursos solo cuando no tiene ninguno, para solicitar nuevos, suelto los que tengo.
Por ejemplo si tengo un proceso que copia datos de una cinta al disco, y luego imprime los resultados en una impresora, con el primer método, solicito la impresora al comienzo, para solo usarla al final, y mientras tanto, nadie la puede usar. Con el segundo método, solicito solo la cinta y el archivo de disco, y cuando termino con los dos, solicito nuevamente el archivo de disco y la impresora.
Las desventajas de los dos protocolos son que si mi proceso utiliza recursos que son muy solicitados, podría esperar indefinidamente a que todos se liberen para usarlos *(inanición)*.

7.4.3 No expropiación

Si un proceso tiene recursos solicitados, y solicita otro, que no se le puede asignar de inmediato, entonces todos los recursos que tiene se liberarán implícitamente. Es aplicable solo a recursos cuyo estado es fácil de guardar.

7.4.4 Espera circular

Se impone un ordenamiento total de los recursos y se exige que cada proceso los solicite en orden creciente de enumeración. Cuando se requieren varios ejemplares de un mismo recurso, se debe emitir una sola solicitud que los incluya a todos. Para pedir recursos con número mayor, debo liberar los que ya tengo.

7.5 Evitación de bloqueos mutuos

La prevención de bloqueos, disminuye el rendimiento del sistema. Se debe pedir información adicional para saber la forma en que se solicitarán los recursos, y el SO debe considerar los recursos disponibles, los ya asignados a cada proceso y las solicitudes y liberaciones futuras de cada proceso para decidir si la solicitud se puede satisfacer o debe esperar.
Una forma es saber el número máximo de recursos de cada tipo que se podría necesitar. El *estado de asignación* de los recursos está definido por el número de recursos disponible y asignado, y las demandas máximas.

7.5.1 Estado seguro

Un estado es *seguro*, si el sistema puede asignar recursos a cada proceso en algún orden, evitando bloqueos. Es seguro si existe una secuencia segura de procesos, si no existe, el estado del sistema es *inseguro*, pudiendo derivar en bloqueo mutuo.
La idea es asegurar que el sistema siempre permanecerá en un estado seguro, asi que antes de asignarle cualquier recurso, debo verificarlo.
Ver ejemplo página 218 del Silberschatz.

7.5.2 Algoritmo de grafo de asignación de recursos

Cuando tenemos un solo ejemplar de cada tipo de recursos, podemos usar una variante del grafo de asignación de recursos. Además de la arista de solicitud y la de asignación, tenemos la de reserva. Va dirigida de P a R e indica que P podría solicitar R en un instante futuro. Se representa con una línea discontinua.
Los recursos deben reservarse a priori en el sistema, o sea que antes de iniciar la ejecución, ya deben aparecer todas las aristas de reserva.
La solicitud de un recurso solo puede ser satisfecha si la conversión de una arista discontinua en continua (asignación de recurso) no da lugar a ciclos.

7.5.3 Algoritmo del banquero

El grafo de asignación, no puede aplicarse a un sistema de asignación de recursos con múltiples ejemplares de cada recurso. Cuando un proceso entra al sistema, debe declarar el número máximo de ejemplares de cada tipo de recursos que podría

necesitar, el cual no puede exceder el total de recursos del sistema.
Cuando el usuario solicita un conjunto de recursos, el sistema debe determinar si la asignación no lo deja en un estado inseguro. Hay que mantener varias estructuras de datos para implementar este algoritmo. Ellas son:

- **Disponible:** Un vector de longitud m indica el número de recursos disponibles de cada tipo. Disponible [j] = k, hay k ejemplares disponibles del recurso R_j.
- **Máx.:** Una matriz n x m define la demanda máxima de cada proceso. Máx. [i, j] = k, el proceso i puede solicitar como máximo k unidades del recurso j.
- **Asignación:** Una matriz n x m define el número de recursos que se han asignado actualmente a cada proceso. Asignación [i, j] = k. El proceso i tiene asignados actualmente k ejemplares del recurso j.
- **Necesidad:** Una matriz n x m indica los recursos que todavía le hacen falta a cada proceso.

Las estructuras de datos anteriores varían en el tiempo, tanto en tamaño como en valor.

7.5.3.1 Algoritmo de seguridad

1. Sean *trabajo* y *fin*, vectores con longitud m y n respectivamente. Inicializo:
   a. Trabajo:= Disponible
   b. Fin[i]:= false

2. Buscar una i tal que:
   a. Fin[i]= false
   b. $Necesidad_i \leq Trabajo$
   c. Si no existe i, continuar con el paso 4

3. Trabajo:= Trabajo + $Asignación_i$
   Fin[i] := true
   Ir al paso 2

4. Si Fin[i] = true para toda i, el sistema esta en estado seguro.

7.5.3.2 Algoritmo de solicitud de recursos

Sea $Solicitud_i$ el vector de solicitudes del proceso P_i. Si $Solicitud_i$ [j] = k, el proceso P_i quiere k ejemplares del tipo de recursos R_j. Para otorgarlos se siguen los siguientes pasos:

5. Si $Solicitud_i \leq Necesidad_i$, ir al paso 2, sino indicar error ya que el proceso excedió su reserva máxima.
6. Si $Solicitud_i \leq Disponible$, ir al paso 3, sino P_i debe esperar a que los recursos estén disponibles.
7. El sistema simula haber asignado los recursos al proceso P_i de la forma:
   $Disponible := Disponible - Solicitud_i$;
   $Asignación_i := Asignación_i + Solicitud_i$;
   $Necesidad_i := Necesidad_i - Solicitud_i$;

Si el estado resultante es seguro, la transacción se lleva a cabo, asignando los recursos a P_i, sino P_i tendrá que esperar, y se restaura el antiguo estado de solicitud de recursos.

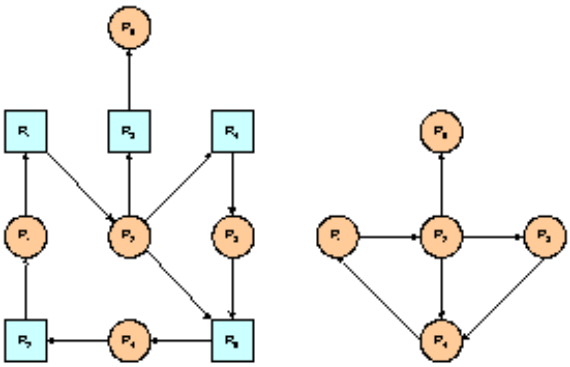
Ver ejemplo 7.5.3.3, página 222 del Silberschatz.

7.6 Detección de bloqueos mutuos

Si el sistema no previene o evita bloqueos mutuos, se puede encontrar con una situación de bloqueo. Para ello necesita:

- Un algoritmo que examine el estado del sistema y determine si hay bloqueo
- Un algoritmo para recuperarse del bloqueo

Este esquema de detección y recuperación requiere un gasto extra que incluye no solo los costos en tiempo para mantener la información sino también para recuperarse después del bloqueo.


Grafo de asignación de recursos

Grafo de espera correspondiente

7.6.1 Un solo ejemplar de cada tipo de recursos

Se utiliza el *grafo de espera*, que se obtiene eliminando del grafo de asignación de recursos, los nodo del tipo de recursos, y uniendo las aristas apropiadas.
Una arista de P_i a P_j implica que el proceso i, espera a que j libere un recurso que i necesita.
Existe un bloqueo mutuo ó en el grafo hay un ciclo. El sistema necesita mantener la información del grafo, y periódicamente invocar el algoritmo de detección de ciclos.

7.6.2 Varios ejemplares de cada tipo de recursos

Para este caso se utilizan estructuras similares a las del algoritmo del banquero.

- **Disponible:** Un vector de longitud m indica el número de recursos disponibles de cada tipo.
- **Asignación:** Una matriz n x m define el número de recursos de cada tipo ya asignados a cada proceso.
- **Solicitud:** Una matriz n x m indica la solicitud actual de cada proceso. Solicitud [i, j] = k à P_i solicita k ejemplares de P_j.

1. Sean *trabajo* y *fin* vectores con longitud m y n respectivamente. Inicializo:
   a. Trabajo:= Disponible
   b. Fin[i]:= false si Asignación[i] <> 0, sino Fin[i]:= true

2. Busco i tal que
   a. Fin[i] = false
   b. $Solicitud_i \leq Trabajo$
   c. Si no existe i, salto a 4

3. Trabajo:= Trabajo + $Asignacion_i$
   Fin[i]:= true
   Ir al paso 2

4. Si Fin[i] = false para alguna i à El sistema está en bloqueo mutuo para ese i.

Ver ejemplo página 226 del Silberschatz.

7.6.3 Uso del algoritmo de detección

Si los bloqueos mutuos son frecuentes el algoritmo de detección deberá invocarse muy seguido ya que los recursos asignados a los procesos bloqueados están ociosos hasta romper ese bloqueo, además el número de procesos bloqueados, puede crecer.

Los bloqueos mutuos solo pueden aparecer cuando algún proceso pide un recurso y no se le puede asignar inmediatamente. Se podría invocar el algoritmo, cada vez que esto sucede, además podemos identificar inmediatamente el proceso causante del bloqueo. Pero invocarlo tan frecuentemente, puede causar demasiado gasto extra de tiempo, por lo tanto se lo puede invocar luego de un tiempo determinado (Ej: 1 hora) o cuando el uso del CPU baja del 40%. En este caso pueden existir muchos ciclos en el grafo, y se dificulta identificar los procesos causantes del bloqueo.

7.7 Recuperación después del bloqueo mutuo

Cuando un algoritmo detecta un bloqueo puede:

- Informar al operador para que lo solucione manualmente
- Dejar que el SO se recupere automáticamente

Las opciones para romper el bloqueo son terminar anormalmente los procesos, o expropiarle recursos.

7.7.1 Terminación de procesos

Hay dos formas:

- **Abortar todos los procesos bloqueados:** El costo es elevado puesto que alguno de los procesos puede haber estado trabajando por largo tiempo.
- **Abortar de a un proceso hasta desbloquearlos:** Incluye un gasto adicional, puesto que luego de eliminar cada proceso hay que llamar nuevamente al algoritmo de detección.

La terminación forzada de procesos no es fácil, puesto que podría dejar incoherencia de datos.
Para saber que procesos del bloqueo se pueden abortar, se debería usar un algoritmo similar al de la planificación de CPU. Se deben abortar los procesos causando el más bajo costo posible.

Los factores para selección de procesos son:

- Prioridad del proceso
- Tiempo que ha trabajado, y cuanto más debe trabajar
- Recursos usados y de que tipo (si los recursos se pueden expropiar fácilmente)
- Recursos adicionales que necesita para terminar su tarea
- Cantidad de procesos para abortar
- Si los procesos son interactivos o por lotes

7.7.2 Expropiación de recursos

Expropiamos recursos de unos procesos para dárselos a otros, hasta romper el bloqueo. Hay que considerar tres aspectos:

- **Selección de la víctima:** Debemos determinar el orden de expropiación para minimizar el costo.
- **Retroceso:** Si expropiamos el recurso, el proceso que lo estaba usando, no puede continuar con su ejecución normal, por lo tanto, el proceso debe retroceder a un estado seguro, y reiniciarlo. La forma más fácil es abortar el proceso y comenzar de nuevo, pero es más conveniente regresar el proceso a un estado seguro, aunque se debe tener guardada información para ello.

- **Inanición:** Podría suceder que siempre se escoja el mismo proceso como víctima, por lo tanto hay que asegurarse que solo se lo va a expropiar un número finito (pequeño) de veces.

7.8 Estrategia combinada para el manejo de bloqueos mutuos

Ninguna de las estrategias anteriores, por si sola es apropiada. Una posibilidad es combinar los tres enfoques básicos y usar el óptimo para cada clase de recursos. Para ello, se pueden ordenar jerárquicamente los recursos, usando dentro de cada clase, la técnica más adecuada, evitando así los bloqueos mutuos.
Clasificación:

- **Recursos internos:** Recursos utilizados por el sistema, como un bloque de control de procesos.
- **Memoria central:** Memoria utilizada por trabajos de usuario.
- **Recursos de trabajos:** Dispositivos asignables y archivos.
- **Espacio intercambiable:** Espacio para cada trabajo de usuario en el almacenamiento auxiliar.

Una solución mixta ordena las clases de la siguiente forma, utilizando las siguientes estrategias:

- **Recursos internos:** Se puede usar prevención mediante ordenamiento de recursos, ya que no es necesario escoger entre solicitudes pendientes en el momento de la ejecución.
- **Memoria central:** Se puede usar prevención mediante expropiación ya que los trabajos pueden intercambiarse a disco, y la memoria central puede expropiarse.
- **Recursos de trabajos:** Se puede usar evitación, ya que la información requerida acerca de las necesidades de recursos, se puede obtener de las tarjetas de control de trabajos.
- **Espacio intercambiable:** Se puede usar preasignación, pues casi siempre se conocen las necesidades de almacenamiento máximas.

8 Gestión de memoria

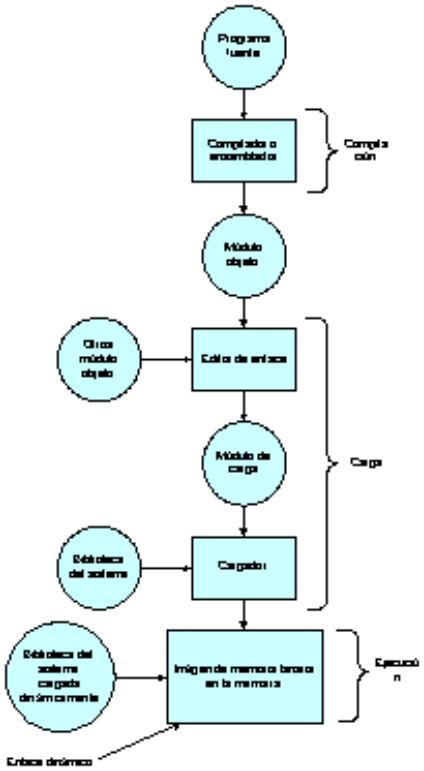
La planificación de la CPU, mejora el rendimiento, pero es necesario mantener varios procesos en memoria, y para eso la memoria se debe compartir.
La selección de un algoritmo de planificación depende principalmente del diseño de hardware del sistema.

8.1 Antecedentes

La memoria es vital para el funcionamiento del sistema. Es una gran matriz de palabras o bytes, cada uno con su propia dirección. La unidad de memoria, solo ve una corriente de direcciones que le llegan, pero no sabe como se generan, ni para que son.

8.1.1 Vinculación de direcciones

Un programa reside en el disco como un archivo binario ejecutable, y para que se ejecute es preciso traerlo a la memoria y colocarlo dentro de un proceso.
Dependiendo de la gestión de memoria, el proceso podría transferirse entre el disco y la memoria durante su ejecución, y la colección de estos procesos que esperan ser transferidos a memoria para ser ejecutados es conocida como *cola de entrada*.
Normalmente se selecciona un proceso de la cola de entrada y se carga en memoria, se ejecuta (pudiendo acceder a instrucciones y datos en memoria), y luego termina, liberando el espacio de memoria que ocupaba (se marca como disponible).



La mayor parte de los sistemas, permiten a un proceso de usuario residir en cualquier parte de la memoria física. Este proceso va a pasar por varias etapas antes de ejecutarse, y las direcciones de memoria, podrían representarse de diferentes maneras en dichas etapas.

La vinculación se puede ejecutar en cualquier etapa del camino:

- **Compilación:** Si en el momento de la compilación se sabe en que parte de la memoria va a residir el proceso, es posible generar código *absoluto*. El código empezará en la posición indicada, pero si la posición de inicio cambia, es necesario recompilar el código.
- **Carga:** Si al compilar, no se sabe en que parte de la memoria va a residir, el compilador debe generar código *reubicable*. La vinculación final se posterga hasta el momento de la carga. Si la dirección de inicio cambia, basta con volver a cargar el código.
- **Ejecución:** Si durante la ejecución, el proceso puede cambiar de un segmento de memoria a otro, la vinculación deberá postergarse hasta el momento de la ejecución. Para esto se requiere hardware especial.

8.1.2 Carga dinámica

El tamaño de un proceso está limitado al tamaño de la memoria física, ya que todos los datos del proceso deben estar en la memoria para que el proceso se ejecute.
La *carga dinámica* aprovecha mejor el espacio de memoria ya que las rutinas no se cargan hasta que son invocadas, y se mantienen en el disco en formato de carga reubicable, por lo tanto, una rutina que no se utiliza, nunca se carga.
La carga dinámica no requiere soporte especial por parte del sistema operativo.

8.1.3 Enlace dinámico

Si no se cuenta con el enlace dinámico, todos los programas del sistema requerirán la inclusión de una copia de las bibliotecas en su imagen ejecutable. Esto desperdicia disco y memoria principal.
Con enlace dinámico, se incluye un *fragmento* (*stub*) en la imagen por cada referencia a una biblioteca, el cual indica como localizar la biblioteca necesaria. Cuando el fragmento se ejecuta, verifica si la rutina está en memoria, y si no está, la carga, logrando una sola copia del código de la biblioteca para todos los procesos.
Si una biblioteca se sustituye por una nueva versión, todos los programas usan la nueva versión. Sin enlace dinámico, se debería volver a enlazar todos los programas con la nueva biblioteca.
Cuando hay distintas versiones de bibliotecas se los conoce como “bibliotecas compartidas”, y cada fragmento, debe guardar información tanto de la biblioteca a cargar, como de la versión.
El enlace dinámico requiere ayuda del SO, ya que para saber si una rutina ya está en memoria, debo comprobar si está en el espacio de memoria de otro proceso, y eso lo administra el SO.

8.1.4 Superposiciones

Para que un proceso pueda ser mayor que la memoria asignada, se usa superposición. Se busca mantener en memoria las instrucciones y datos necesarios. Si se requieren nuevos, se borran los que ya no se necesitan.

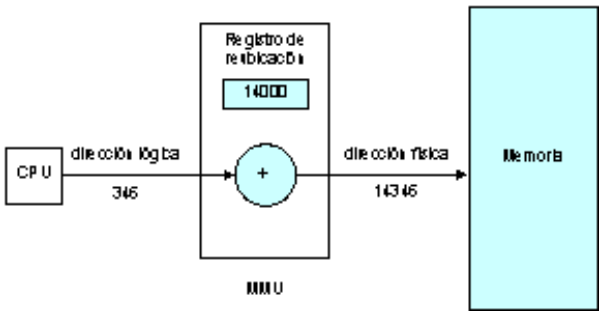
La superposición no requiere soporte por parte del SO.

8.2 Espacio de direcciones lógico y físico

Una dirección generada por la CPU, se denomina *dirección lógica*, mientras que otra percibida por la unidad de memoria, se conoce como *dirección física*.
La vinculación de direcciones en la compilación o carga da pie a que las direcciones lógicas coincidan con las físicas. En cambio durante la ejecución, las direcciones difieren. En este caso, la dirección lógica es conocida como "*dirección virtual*".

La transformación de direcciones virtuales a físicas en el momento de ejecución corre por cuenta de la *unidad de gestión de memoria* (MMU) que es un dispositivo de hardware.

El registro base, es ahora el registro de *reubicación*, y el valor que contiene se suma a todas las direcciones generadas por los procesos de usuario. MS-DOS utiliza cuatro registros de reubicación al ejecutar procesos. El programa de usuario nunca ve las direcciones físicas reales.



8.3 Intercambio

Un proceso tiene que estar en la memoria para ser ejecutado, pero podría pasar temporalmente a un almacenamiento auxiliar, y luego traerse a la memoria nuevamente.
Esto se puede usar para planificación de CPU. Por ejemplo en Round Robin cuando se termina el cuanto, el proceso que se estaba ejecutando es colocado en el disco. De la misma forma cuando se está ejecutando un proceso de prioridad X, y llega otro con mayor prioridad.

Normalmente cuando el proceso fue al disco, vuelve al mismo espacio de memoria donde estaba inicialmente. Esto depende de cuando se realice la vinculación de direcciones. Si se hace durante la carga o compilación, debe volver al mismo lugar. Si es durante la ejecución, el proceso se puede reubicar, ya que calculo las direcciones al ejecutarlo.

En el almacenamiento auxiliar (disco) se guardan las imágenes de los procesos, y el sistema mantiene la *cola de procesos listos*, guardando datos de donde se encuentran dichas imágenes.

Cuando el planificador de CPU llama al despachador, éste verifica si el proceso está en memoria. Si no es así, verifica si hay memoria suficiente para cargarlo, y si no hay, guarda en el disco otro proceso y trae al que va a ejecutar. El tiempo de conmutación de contexto es alto, y depende de la cantidad de memoria que se intercambie, además, si queremos intercambiar un proceso, hay que verificar que está totalmente inactivo, teniendo en cuenta principalmente que no tenga E/S pendiente.

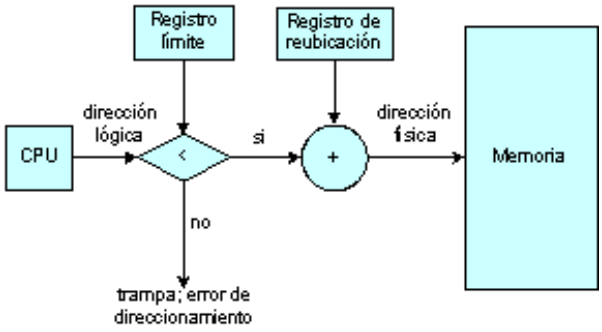
En general, el área de SWAP, se asigna como un área del disco independiente, a fin de usarla lo más rápido posible.
En UNIX se usa una modificación de este intercambio. Normalmente está desactivada y se inicia si hay demasiados procesos en ejecución.

8.4 Asignación contigua

La memoria principal debe dar cabida al SO y a los procesos. Generalmente se divide en dos particiones, una para cada uno. Es posible colocar al SO en memoria alta o baja, y esta decisión se toma en base a la ubicación del vector de interrupciones. Ya que se encuentra en memoria baja, se trata de colocar el SO en la misma área.

8.4.1 Asignación con una sola partición

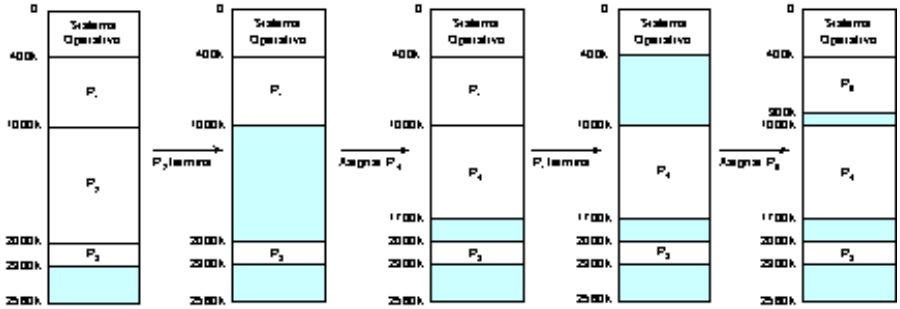
Si el sistema se encuentra en memoria baja y los procesos de usuario en memoria alta, hay que proteger el código del SO para que los procesos no lo sobre escriban, y hay que proteger los procesos de usuario, uno del otro.
Si se usan registros de reubicación, y registros límites, toda dirección lógica debe ser menor que el registro límite.



La MMU dinámicamente transforma la dirección lógica en física (usando el registro de reubicación). La dirección es válida, ya que la CPU la coteja con los registros. De esta forma se puede proteger al SO y los procesos.

8.4.2 Asignación con múltiples particiones

Hay que repartir la memoria disponible entre los distintos procesos de usuario que están en la cola de entrada, esperando a ser transferidos a la memoria.
Uno de los esquemas más sencillos, es dividir la memoria en *particiones* de tamaño fijo, donde cada una contiene un único proceso, y el grado de multiprogramación está limitado por el número de particiones. Ese esquema ya no se usa.
Otro esquema consiste en que el SO mantiene una tabla que indica que partes de la memoria están disponibles, y cuales no.
Cuando un proceso llega a memoria se busca un *hueco* de tamaño suficiente para colocarlo. Si se halla, se asigna solo la memoria necesaria, y el resto se deja disponible.
En cualquier instante dado tenemos una lista de los tamaños de bloque disponibles, y la cola de entrada, pudiendo el planificador esperar a que un proceso termine, y libere memoria para ejecutar el próximo proceso de la cola, o buscar en la cola un proceso más pequeño que entre en los huecos existentes.



Hay varias estrategias para elegir huecos:

- **Primer ajuste:** Asigno el primer hueco con tamaño suficiente. Se puede comenzar la búsqueda desde varios puntos.
- **Mejor ajuste:** Asigno el hueco más pequeño que tiene tamaño suficiente. Es preciso examinar toda la lista, a menos que esté ordenada por tamaño. Produce el hueco remanente más pequeño.
- **Peor ajuste:** Asigno el hueco más grande. Hay que examinar toda la lista, y se produce el hueco remanente más grande, que en muchos casos podría ser más útil que el más pequeño.

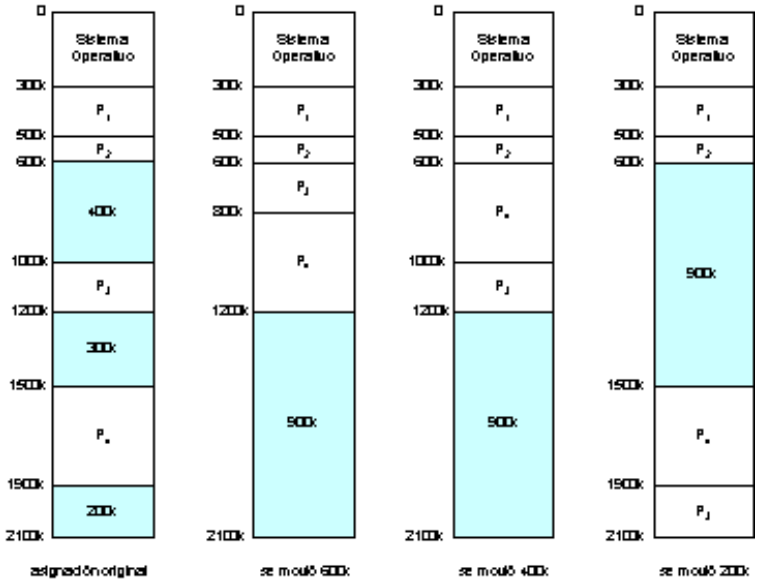
Se ha demostrado que el primer ajuste y el mejor ajuste son mejores que el peor ajuste en términos tanto de tiempo como de aprovechamiento de memoria. El primer ajuste suele ser más rápido.

8.4.3 Fragmentación externa e interna

Hay *fragmentación externa* cuando se cuenta con suficiente espacio total para satisfacer la solicitud, pero el espacio no es contiguo, está dividido en huecos pequeños.
Esta fragmentación puede ser grave, ya que si la memoria estuviera en un solo bloque, se podrían ejecutar más procesos.

Otro problema es cuando se asignan múltiples particiones. El proceso es más pequeño que la partición, por lo tanto en esa partición queda memoria inutilizada. Esto es llamado *fragmentación interna*.

Una solución al problema de fragmentación externa es la *compactación*. Se desplaza el contenido de memoria, hasta tener toda la memoria libre en un solo bloque. El problema es que solo es posible si la reubicación es dinámica, y en el momento de ejecución. Y el costo depende del algoritmo.
El algoritmo más simple es desplazar todos los procesos hacia un lado, quedando el hueco en el otro, pero puede ser costoso.
El intercambio también puede combinarse con compactación.



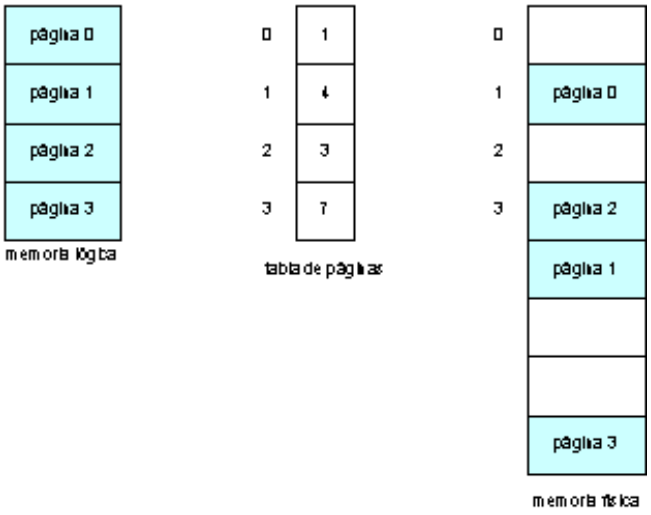
8.5 Paginación

Otra posible solución al problema de fragmentación externa es permitir que el espacio de direcciones lógicas de un proceso no sea contiguo. De esta forma podemos asignar memoria siempre que haya disponible.

8.5.1 Método básico

La memoria física se divide en bloques de tamaño fijo llamado *marcos*, y la memoria lógica también lo hace en *páginas*.
Cuando se va a ejecutar un proceso, se cargan sus páginas desde el almacenamiento auxiliar, a cualquier marco disponible. El almacenamiento auxiliar se divide en bloques de tamaño fijo, que tienen el mismo tamaño que los marcos de memoria.
Para hacerlo se debe contar con soporte de hardware.

Cada dirección generada por CPU se divide en dos partes: un *número de página* (p) y un *desplazamiento de página* (d).
El número de página se utiliza como índice de una tabla de páginas, la cual contiene la dirección base de cada página en la memoria física.
Esta dirección base, combinada con el desplazamiento, da la dirección física que estábamos buscando.
El tamaño de la página está definido por el hardware.
La paginación es una forma de reubicación dinámica. Se vincula cada dirección lógica con una física, y no permite la fragmentación externa.
Cualquier marco libre, se puede asignar. Lo que si puede existir es la fragmentación interna.



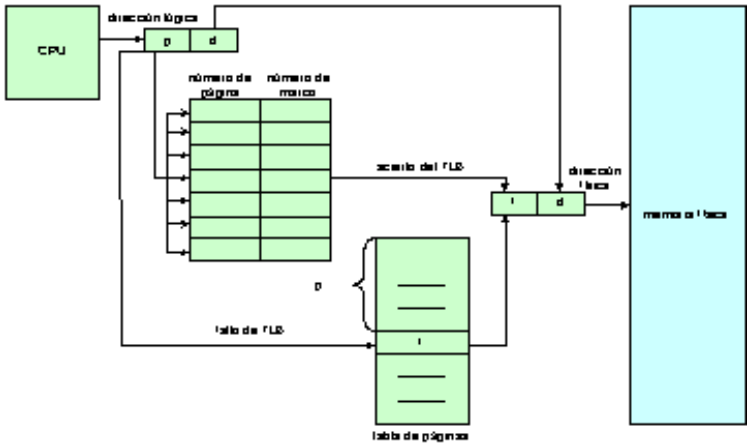
Si el tamaño de las páginas fuera pequeño, disminuiría la fragmentación interna, pero tendríamos un gasto extra para guardar cada entrada a página. Además, la E/S es más eficiente cuando la cantidad de datos es mayor. Hoy en día, las páginas son entre 2 y 4 Kb.

Cuando un proceso ingresa al sistema para ejecutarse, se calcula su tamaño expresado en páginas, y se ve si hay suficientes marcos libres para almacenarlo.
El programa de usuario, ve la memoria como un espacio contiguo, aunque en realidad está disperso por toda la memoria física.
El SO administra la memoria, guardando información en la *tabla de marcos* que indica si el marco está libre, o asignado, y en caso último, quien lo tiene. Además tiene una copia de las tablas de páginas de cada proceso, para traducir direcciones lógicas en físicas que también es utilizada por el despachador de CPU, por lo tanto la paginación aumenta el tiempo de conmutación de contexto.

8.5.2 Estructura de la tabla de páginas

Cada SO tiene un método propio de almacenamiento de páginas, y casi todos asignan una tabla de páginas para cada proceso.

8.5.2.1 Soporte de hardware



La tabla se implementa como un conjunto de registros dedicado, construidos con circuitos lógicos de alta velocidad, a fin de hacer eficiente la traducción de direcciones, ya que todo acceso a memoria, debe pasar por el mapa de paginación. Las instrucciones para cargar o modificar los registros de la tabla de páginas son privilegiadas.
El sistema de registros es bueno si la tabla es pequeña, sino, la tabla se mantiene en memoria principal, y se guarda un *registro base de tablas de página* que apunta a ella.
El problema de este enfoque es el tiempo requerido para acceder a memoria del usuario, ya que se requieren dos accesos a memoria para acceder a un byte.
La solución standard es usar un pequeño caché que guarda las direcciones usadas recientemente. Si el número buscado no está en el registro, se accede a memoria y se lo guarda para una futura búsqueda.

El porcentaje de veces que un número de página se encuentra en el caché (*registros asociativos*) es llamada *tasa de aciertos*. El *tiempo de acceso efectivo*, contempla los dos casos, si la página se encuentra o no en el caché, y multiplica los diferentes tiempos de acceso, por la tasa de aciertos, de la forma:

tiempo de acceso efectivo = ta x (bc + bm) x (1 – ta) x (bc + bm + bm)

ta = tasa de aciertos
bc = tiempo de búsqueda en el caché (acceso al caché)
bm = tiempo de búsqueda en la memoria (acceso a la memoria)

8.5.2.2 Protección

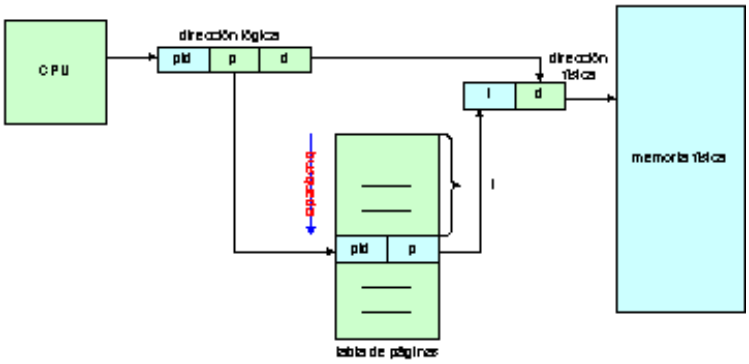
Se logra con bits de protección asociados a cada marco, normalmente mantenidos en la tabla de páginas. Pueden indicar si una página es de lectura-escritura o solo lectura, y al estar en la tabla, se pueden controlar cada vez que se intenta acceder a memoria.
Se añade a la tabla un bit de *validez/no validez* indicando del espacio de direcciones del proceso, que páginas están cargadas en memoria y cuales no.
Un proceso casi nunca utiliza todo su espacio de direcciones, por lo tanto sería un desperdicio crear una tabla de páginas con entradas de cada una de las direcciones.
Algunos sistemas cuentan con hardware en forma de *registro de longitud de la tabla de páginas* que indica el tamaño de la tabla de páginas, cotejando si la página pedida se encuentra o no en las utilizadas por el proceso.

8.5.3 Paginación multinivel

En los sistemas modernos, la tabla de páginas crece demasiado, por lo tanto queremos dividirla en fragmentos más pequeños, aunque en un sistema con un espacio de direcciones lógico de 64 bits, la paginación en niveles ya no es apropiada.
Cálculo de tamaño de páginas: Ver página 267 Silberschatz.
De todas formas, si tengo por ejemplo una paginación de 3 niveles, voy a necesitar 4 accesos a memoria para obtener el dato que necesito.

8.5.4 Tabla de páginas invertida

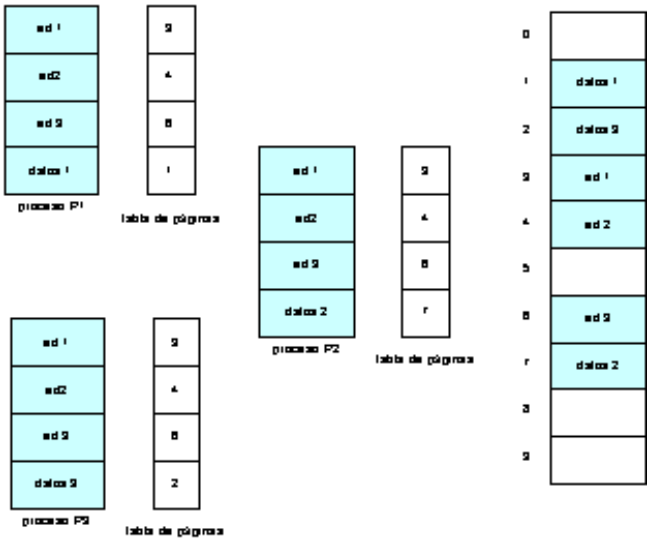
Cada proceso tiene su tabla de páginas, conteniendo entradas por cada página que el proceso usa. De esta forma, cada tabla podría contener millones de entradas, ocupando mucha memoria, y para resolverlo, se usa una *tabla de páginas invertida*.
Tiene una entrada por cada marco real de la memoria, y contiene información sobre el proceso dueño, por lo tanto hay una sola tabla de páginas en el sistema, conteniendo una única entrada por cada página de memoria física.



Este esquema reduce la cantidad de memoria necesaria para almacenar tablas, pero aumenta el tiempo de búsqueda, ya que está ordenada por dirección física, y nosotros buscamos la lógica.

8.5.5 Páginas compartidas

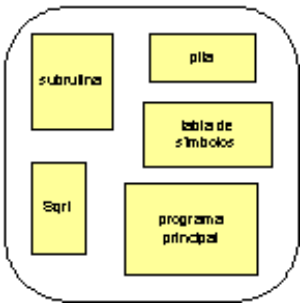
Se puede compartir código común. Ej: Si tengo un sistema con 40 usuarios usando el mismo editor de texto, sería útil que el código del ese editor se pudiera compartir.
Para ser compartible, el código debe ser *reentrante*, o sea, que no puede modificarse a si mismo.
La tabla de páginas de cada usuario, establece una correspondencia con la misma copia física del editor, pero las páginas de datos corresponden a marcos distintos.
Los sistemas que utilizan páginas invertidas tienen problemas para implementar esta memoria compartida.



8.6 Segmentación

La visión del usuario no es igual a la memoria física real; sólo tiene una correspondencia con ella.

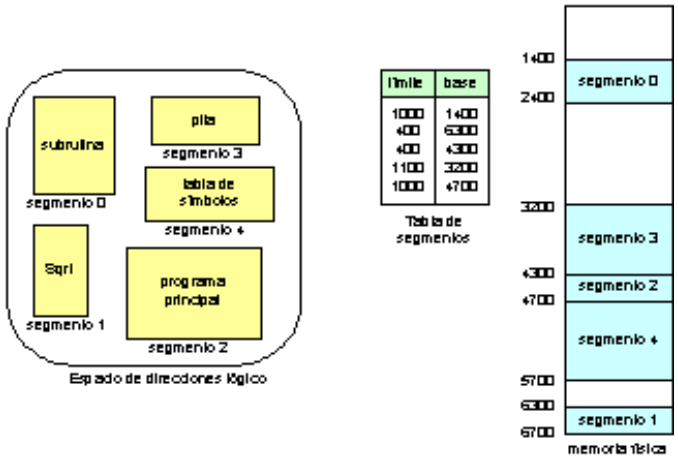
8.6.1 Método básico



El usuario ve la memoria como una colección de segmentos de tamaño variable, sin ordenamiento entre ellos. Lo vemos como un programa principal, con subrutinas, procedimientos, funciones, y estructuras de datos.
La *segmentación*, es un esquema de gestión de memoria, que apoya esta visión del usuario. Cada dirección tiene ahora dos cantidades. El número de segmento (sustituyendo los nombres) y el desplazamiento.
Por ejemplo se podrían crear los diferentes segmentos en un programa Pascal:

1. Variables globales
2. Pila de llamadas a procedimientos
3. Código de cada procedimiento
4. Variables locales de cada procedimiento

8.6.2 Hardware



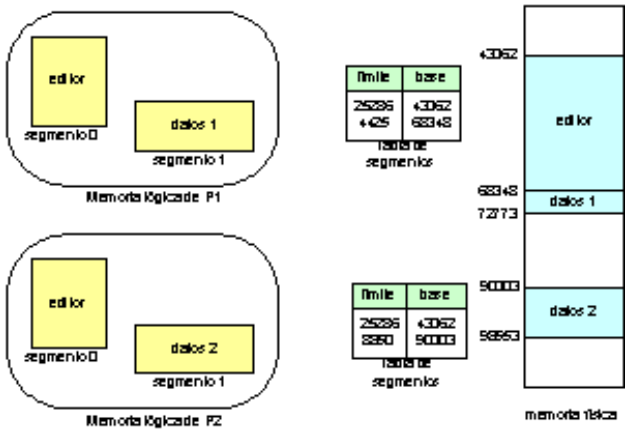
El usuario usa un direccionamiento bidireccional, pero la memoria es unidireccional, por lo tanto hay que definir una *tabla de segmentos* que transforma direcciones.

Cada entrada de la tabla tiene una base del segmento (dirección física inicial) y un límite (longitud del segmento).

8.6.3 Implementación de las tablas de segmentos

Al igual que con paginación, se puede guardar la tabla en registros, o si es grande, en memoria principal. Se maneja de la misma forma.

8.6.4 Protección y compartimiento



Podemos asociar la protección con los segmentos. Algunos segmentos contienen instrucciones (que no se modifican a si mismas), y otros contienen datos, así que los segmentos de instrucciones, pueden ser de solo lectura.
El hardware de transformación examina los bits de protección, evitando accesos no permitidos a memoria, chequeando también que los índices de acceso no sobrepasen los límites del segmento.
El compartimiento se hace a nivel de segmentos. Y cualquier información compartible se define como un segmento.
En estos casos se genera un problema, cuando los segmentos contienen referencias a si mismos (por ejemplo saltos). En ese caso, todos los programas que utilicen ese segmento, lo deben numerar de la misma forma, por ejemplo, para todos debe ser el segmento 4. Se hace cada vez más difícil encontrar un número de segmento adecuado, a medida que aumenta el número de programas que lo utilizan.

8.5.6 Fragmentación

Los segmentos tienen longitud variable, por lo tanto su ubicación se decide dinámicamente, por los algoritmos de mejor o peor ajuste, pudiendo causar fragmentación externa. Como el algoritmo de reubicación es de naturaleza dinámica, se puede compactar la memoria en cualquier momento.

8.7 Segmentación con paginación

Se busca combinarlos para mejorarlos.

8.7.1 MULTICS

Una dirección lógica se forma con un número de segmento de 18 bits, y un desplazamiento de 16 bits (32 bits total). Se crea un gasto extra para mantener la tabla de segmentos, pero es tolerable, ya que solo se tiene entradas por los segmentos que se poseen (no hay entradas vacías).
Debido al tamaño del segmento, la fragmentación externa, podría ser un problema, o el tiempo de búsqueda para mejor o peor ajuste, muy largo.
La solución fue *paginar los segmentos*. Limitando la fragmentación externa, y facilitando la asignación de memoria (se asigna el primer marco disponible). El desplazamiento dentro del segmento se divide en el número de página (16 bits) y la distancia en la página (10 bits).
La entrada segmentable no contiene la dirección base del segmento, sino la dirección base de una tabla de páginas para ese segmento.
Tenemos una tabla individual para cada segmento, pero de un tamaño tolerable. Eliminamos la fragmentación externa, pero

tenemos interna, además de aumentar el gasto en espacios para tablas.
Cuando la tabla de segmentos es grande, también se pagina.

8.7.2 Versión de 32 bits de OS/2

El número máximo de segmentos por proceso es de 16 k, con un tamaño de hasta 4 Gb por segmento. El tamaño de páginas es de 4 Kb.
El espacio de direcciones lógico, se divide en dos particiones. Una con 8k en segmentos privados, y otra con 8k de segmentos compartidos con otros procesos. La información sobre la primera partición se mantiene en la *tabla de descriptores locales*, y sobre la segunda en la *tabla de descriptores globales*.
Para mejorar la eficiencia del uso de memoria, las páginas del Intel 386, se pueden pasar a disco.

9 Memoria Virtual

La memoria virtual es una técnica que permite ejecutar procesos que podrían no estar totalmente en la memoria, por lo tanto los programas pueden ser más grandes que la memoria física.
Hace una abstracción de la memoria considerándola como una matriz uniforme y extremadamente grande de almacenamiento, separando la memoria lógica que el usuario ve de la memoria física, y evitando que los programadores tengan que preocuparse por las limitaciones.
Es difícil de implementar, y si no se usa con cuidado puede reducir sustancialmente el rendimiento.

9.1 Antecedentes

La memoria virtual generalmente se implementa con *paginación por demanda*, pero también puede implementarse en un sistema con segmentación. Varios sistemas ofrecen un esquema de segmentación paginada, en los que los segmentos se dividen en páginas. También es posible usar *segmentación por demanda*, el problema es que son algoritmos más complejos porque los segmentos son de tamaño variable.

9.2 Paginación por Demanda

Un sistema de paginación por demanda es similar a un sistema de paginación con intercambio. Los procesos residen en memoria secundaria, y cuando queremos ejecutarlo lo pasamos por intercambio a la memoria, pero en vez de intercambiar todo el proceso, usamos un intercambio *perezoso* (nunca intercambia una pagina a la memoria si no se va a necesitar).
Un intercambiador intercambia todo un proceso, por lo cual a nuestro intercambiador perezoso lo denominamos *paginador*.
Cuando un proceso se va a cargar en memoria, el paginador adivina cuales son las páginas que se usaran antes de que el programa se pase nuevamente al disco. Notar que así se reduce el tiempo de carga y la memoria física requerida.
Es necesario el soporte de hardware, para poder distinguir entre las páginas que están en memoria y las paginas que están en disco. Esto se puede lograr con el bit de *validez-no validez*, valido indicara que la pagina esta en la memoria (y es válida, o sea, está dentro de las direcciones del proceso), no valida indicara que no es válida, o no está en memoria.
Si adivinamos correctamente cuales son las paginas que el proceso va a utilizar, y solo cargamos esas, entonces el programa ejecutara normalmente como si hubiese sido cargado totalmente. Si el proceso intenta acceder a una página no valida (no es error de direccionamiento), causa una trampa de *fallo de pagina (page fault)* y le transfiere el control al SO y ejecuta el siguiente procedimiento:

1. Se consulta una tabla interna (se encuentra usualmente en el PCB del proceso) para ver si la referencia es un acceso valido o invalido a la memoria.
2. Si la referencia no es valida, se termina el proceso. Si la referencia es valida pero la página no se había cargado, es cargada.
3. Se busca un marco libre donde cargar la página.

4. Se planifica una operación de disco para leer la página deseada y colocarla en el marco libre.
5. Al termina la lectura, se modifica la tabla consultada en 1 poniendo que la página ya es valida (esta cargada).
6. Reiniciamos la instrucción que se interrumpió por la trampa de dirección no valida. Ahora el proceso puede acceder a la página como si nada hubiese pasado.

Notar que como se guarda el estado (registros, código de condición, contador de instrucciones) del proceso interrumpido al ocurrir la falla de página, se puede reiniciar el proceso exactamente en el mismo lugar y estado.
En el caso extremo podríamos iniciar la ejecución de un proceso sin páginas en la memoria, y al ir accediendo a las instrucciones, se irían cargando las páginas necesarias. Este esquema se llama *paginación por demanda pura*. Nunca se carga una página si no se necesita.
La idea de que los programas necesiten tener en memoria varias páginas para ejecutar una instrucción (generando varios fallos de página), es improbable.

El hardware para apoyar la paginación por demanda es el mismo que apoya la paginación y el intercambio:

- **Tabla de Páginas:** Marca las entradas válidas y no válidas con el bit de *validez- no validez*.
- **Memoria Secundaria:** Contiene las páginas que no están en memoria principal.

Además de esto se requiere una buena cantidad de software, y algunas restricciones arquitectónicas como la necesidad de poder reiniciar una instrucción después de un fallo de página. Y como peor caso, si el fallo ocurrió mientras se está obteniendo un operando, es preciso volver a obtener la instrucción y decodificarla. Ver ejemplo página 296 del Silberschatz.

El principal problema surge cuando una instrucción puede modificar varias posiciones distintas, y esas posiciones se traslapan. Por ejemplo, mover caracteres de un bloque a otro (y los dos se traslapan). Si ocurre un fallo en el medio, se perdieron datos que fueron sustituidos.
Hay 2 formas de resolver el problema:

- Se intenta acceder a los dos extremos de los bloques de memoria a modificar y si ocurre un fallo cargar la página antes de ejecutar la modificación.
- Guardar en registros temporales los valores sobrescritos y si ocurre un fallo volver hacia tras antes de activar la trampa.

Es posible usar un esquema de paginación en cualquier sistema, mientras no sea de paginación por demanda.

9.3 Desempeño de la Paginación por Demanda

La paginación por demanda puede tener un efecto importante sobre el desempeño de un sistema. Para ver el efecto debemos calcular el *tiempo de acceso efectivo* a la memoria paginada por demanda.
Sea p la probabilidad de que ocurra un fallo (p es muy cercano a 0) y am el tiempo de acceso a memoria (que varia entre 10 y 200 nanosegundos), entonces:

Tiempo de acceso efectivo = (1 – p) x am + p x tiempo de fallo de pagina

Para calcular el tiempo de fallo de página, debemos tener en consideración, varios tiempos. Los tres principales son:

1. Atención de la interrupción de fallo de página.
2. Traer la página a memoria.
3. Reiniciar el proceso.

Entonces, el tiempo de acceso efectivo es directamente proporcional a la frecuencia de fallos. Esto denota la importancia de mantener baja la frecuencia de fallos.
Ver ejemplo página 299 Silberschatz.

Otro aspecto es el manejo y uso general del espacio de intercambio. La E/S de disco al espacio de intercambio generalmente es más rápida que al sistema de archivos, por lo que es posible mejorar el rendimiento de la paginación copiando una imagen de archivo completa en el espacio de intercambio al arrancar el proceso y luego realizar la paginación por demanda usando el espacio de intercambio.
Los sistemas con espacio de intercambio limitado pueden utilizar un esquema distinto si emplean archivos binarios. Las páginas solicitadas se traen directamente del sistema de archivos. Si llega a ser necesario el reemplazo de páginas en la memoria, estas páginas pueden simplemente sobrescribirse y leerse de nuevo del sistema de archivo si se necesitan.
Una tercera opción es solicitar inicialmente las páginas al sistema de archivos pero escribirlas en el espacio de intercambio cuando se reemplazan. Esto asegura que solo se traerán del sistema de archivos las páginas necesarias y que toda la paginación subsiguiente se realizara desde el espacio de intercambio.

9.4 Reemplazo de Páginas

En los esquemas vistos la frecuencia de fallos de página no han representado un problema grave. Pensemos en que un proceso que requiere 10 páginas, generalmente necesita 5 de ellas cargadas. Con este esquema ahorramos la E/S de traer las 10 páginas a memoria, además se puede ampliar la multiprogramación ejecutando el doble de procesos (ya que cargo la mitad de las páginas).
Si uno de estos procesos que usa 5 paginas produce un fallo de pagina (requiere de una sexta pagina), ocurre una trampa y se transfiere el control al SO. No quedan marcos libres, toda la memoria está ocupada. Al llegar a este estado, el SO tiene varias opciones:

- **Terminar el proceso de usuario.** Esta opción no es muy aceptable debido a que el SO hace paginación por demanda para aumentar la productividad del computador, y además este esquema debe ser transparente para el usuario.
- **Intercambiar a disco un proceso.** Haciendo esto se liberan algunos marcos, pero se reduce el nivel de multiprogramación. Esta opción es bastante buena pero hay otra más interesante.
- **Reemplazo de páginas.** Si no hay marcos libres se busca uno que no se este usando y lo liberamos.

La liberación de un marco es posible hacerla escribiendo su contenido en el espacio de intercambio y modificando la tabla de páginas (y las demás tablas) de modo que indiquen que la página ya no esta en la memoria. Ahora el marco esta libre y puede contener la pagina por la cual ocurrió un fallo.
Entonces, debemos modificar la rutina de servicio de fallos de página (capítulo 9.2) de la siguiente forma:

```
Si hay un marco libre
   Se usa;
Sino
{
   Se usa un algoritmo de reemplazo de paginas para escoger el marco victima;
   Escribir la pagina victima en el disco y modificar las tablas de paginas y marcos;
   Leer la página deseada y colocarla en al marco liberado, modificar tablas;
}
Fin Si
```

Si no hay un marco libre, se requieren 2 transferencias de páginas (una hacia fuera y una hacia adentro), duplicando el tiempo de servicio de fallo de página. Este gasto extra puede ser solucionado empleando un *bit de modificación* o *bit sucio*.
Cada página o marco puede tener en hardware un bit asociado. El cual es encendido si se escribe una palabra o byte en la página (indicando que se modifico). De esta forma evitamos la escritura de la pagina victima si no ha sido modificada.
El reemplazo de páginas es básico para la paginación por demanda, pues hace la total separación entre la memoria lógica y física, y ofrece a los programadores una memoria virtual muy grande con una memoria física muy pequeña.

Para implementar la paginación por demanda, entonces debemos resolver 2 problemas:

- **Un algoritmo de asignación de marcos.** Es necesario decidir cuantos marcos se asignan a cada uno.
- **Un algoritmo de reemplazo de páginas.** Hay que seleccionar que página se reemplaza cuando se necesario.

Estos algoritmos deben ser bien diseñados para ganar un buen desempeño del computador.

9.5 Algoritmo de Reemplazo de Páginas

Hay muchos, y es probable que cada SO tenga su propio esquema de reemplazos. Lo que queremos es el algoritmo con *frecuencia de fallos de página* más baja.
La evaluación de los algoritmos se hace ejecutándolo con una serie específica de referencias a la memoria (denominada *serie de referencias*) y calculando el número de fallos de página. Además se debe saber el número de marcos disponibles en el sistema (cuantos más hallan menos fallos ocurrirán).

9.5.1 Algoritmo FIFO

Es el más sencillo. Este algoritmo asocia a cada página el instante en que se trajo a la memoria. Cuando es necesario reemplazar una página se escoge la más vieja. Esto se puede implementar con una cola simple.
Su desempeño no es siempre bueno, ya que la pagina más vieja puede contener una variable que se esta consultando constantemente.
Un problema importante de este algoritmo se conoce como *anomalía de Belady*. Consiste en que al aumentar la cantidad de marcos pueden aumentar la cantidad de fallos para algunas series de referencias.

9.5.2 Algoritmo óptimo

Este algoritmo reemplaza la página que no se **usará** durante más tiempo, asegurando la frecuencia de fallos más baja para un número fijo de marcos.
El problema es que es difícil de implementar porque requiere un conocimiento futuro de la serie de referencias.

9.5.3 Algoritmo LRU

Least recently used, es el algoritmo que reemplaza la página que no se ha usado por más tiempo. Asocia a cada página el instante en que se uso por ultima vez, y así selecciona la que hace más tiempo que no se usa.
Esta política se usa mucho y se considera excelente, el problema es como implementarla, ya que requiere ayuda sustancial del hardware.

Hay 2 implementaciones posibles:

- **Contadores:** Asociamos a cada entrada de la tabla de páginas el campo de “tiempo de uso” y añadimos al CPU un reloj lógico o contador. El reloj se incrementa en cada referencia a la memoria. Cada vez que se hace referencia a una página, el contenido del reloj se copia en el campo de tiempo de uso. Luego hay que hacer una búsqueda para encontrar la página. Otro problema a resolver es el overflow del reloj.
- **Pila:** La estrategia es mantener una pila de números de página. Cuando se hace referencia a una se saca de la pila y se coloca arriba. Así en el tope esta la más reciente y en la base la LRU. La pila se debe implementar como una lista doblemente enlazada. Cada actualización es un poco más costosa, pero ya no es necesario buscar para reemplazar. Este enfoque es el más apropiado para implementaciones en software.

9.5.4 Algoritmos de Aproximación a LRU

Pocos sistemas de computación ofrecen suficiente soporte de hardware para el verdadero reemplazo de paginas LRU. Los que

no cuentan con apoyo de hardware no pueden hacer otra cosa que usar el FIFO. Los que cuentan con un bit que el hardware enciende cada vez que se accede a una página, (permitiendo saber que páginas se usaron y cuales no), pueden utilizar algoritmos que se aproximan a LRU.

9.5.4.1 Algoritmo con Bits de Referencias Adicionales

Mantenido un byte histórico para cada página de una tabla que esta en memoria. A intervalos regulares una interrupción ejecuta una rutina del SO que desplaza un bit a la derecha en el byte histórico y pone el bit de referencia de la página en el bit más significativo (del byte histórico). Así la página con número más bajo será la elegida, y como el número puede no ser único se puede eliminar diferencias con FIFO u otro algoritmo.

9.5.4.2 Algoritmo de Segunda Oportunidad

El algoritmo es igual a FIFO, pero cuando se va a elegir una página se mira el bit de referencia, si es 0 se reemplaza, sino se le da una segunda oportunidad (y se pone el bit en 0), mirando a continuación la siguiente en orden FIFO.
La implementación es generalmente mediante una cola circular.

9.5.4.3 Algoritmo de Segunda Oportunidad Mejorado

Consiste en considerar tanto el bit de referencia como el bit de modificación como un par ordenado. Obteniendo los 4 siguientes casos:

- **(0, 0).** Ni se uso ni se modifico, es la mejor para reemplazar.
- **(0, 1).** No se uso pero se modifico, hay que escribirla en disco antes de reemplazarla por lo cual no es tan buena.
- **(1, 0).** Se uso pero no se modifico, es posible que se use pronto.
- **(1, 1).** Se uso y se modifico, se va a usar probablemente pronto y será necesario escribirla antes de reemplazarla.

Este esquema es utilizado por Macintosh.

9.5.5 Algoritmos de Conteo

Manteniendo un contador de número de referencias que se han hecho a cada página se puede hacer:

- **Algoritmo LFU.** Se elige el menos frecuentemente usado (*least frecuently used)*, el de cuenta más baja. No es eficiente en el caso de páginas que se usaron mucho al comienzo del proceso, y luego no se volvieron a usar (tienen cuenta alta).
- **Algoritmo MFU.** Se elige la más frecuentemente usada. Se basa en el argumento de que la última página traída a memoria, es la que tiene cuenta más baja.

Estos algoritmos no son comunes y poco se aproximan al ÓPTIMO.

9.5.6 Algoritmo de Colocación de Páginas en Buffers

Muchos sistemas mantienen una *reserva* de marcos libre. Cuando ocurre un fallo de pagina se escoge un marco víctima igual que antes, pero, la página deseada se coloca en un marco libre de la reserva antes de escribir la víctima en disco. Este procedimiento permite al proceso reiniciarse lo más pronto posible, sin esperar que la victima se escriba en el disco. Cuando termino de escribir la víctima, añado su marco a la reserva de marcos libres.
También se puede usar esta idea, para ir escribiendo en disco las páginas modificadas, así, cuando son elegidas para ser reemplazadas están “límpias”.
Otra forma, es manteniendo una reserva de marcos libres, pero recordando que página estaba en cada uno, así, si fue escrita a disco, reemplazada, y la volvemos a necesitar, todavía se puede encontrar en el marco libre, sin ser sobrescrita.

9.6 Asignación de Marcos

Uno de los grandes problemas es como saber cuantos marcos hay que darle a cada proceso, sobretodo cuando el sistema es multiprogramado (hay varios procesos en memoria a la vez).

9.6.1 Numero Mínimo de Marcos

El número máximo esta definido por la cantidad de memoria física disponible. Como vimos, cuantos menos marcos asignemos la frecuencia de fallos de página aumenta, por eso hay un número mínimo de marcos que es preciso asignar y esta definido por la arquitectura del conjunto de instrucciones. Es necesario tener suficientes marcos para contener todas las páginas a las que una sola instrucción puede hacer referencia.
La situación más grave ocurre en arquitecturas que permiten múltiples niveles de indirección, por lo tanto, una página podría tener referencia a otra, y ésta a otra, y así sucesivamente, necesitando tener cargadas todas las páginas en memoria para poder ejecutar una simple instrucción.
Por esta causa, se limita el nivel de indirección asignando como máximo, la cantidad de marcos disponibles en la memoria física.

9.6.2 Algoritmos de Asignación

- **Asignación Equitativa.** Se dividen los m marcos entre los n procesos, dando a cada uno una porción equitativa de la cantidad de marcos.
- **Asignación Proporcional.** Se le asigna memoria disponible a cada proceso según su tamaño.

Tanto en la equitativa como en la proporcional la asignación a cada proceso podría variar según el nivel de multiprogramación (cuantos más procesos haya en memoria, menor es la cantidad de marcos asignados a cada uno). Además, en ambos casos se trata igual a un proceso de alta prioridad que a un proceso de baja prioridad. Por esto, otras estrategias ofrecen marcos dependiendo del nivel de prioridad, o según una combinación de tamaño y prioridad del proceso.

9.6.3 Asignación Global o Local

Podemos clasificar los algoritmos de reemplazo de páginas en dos categorías:

- **Reemplazo Global.** Permite seleccionar la víctima de todos los marcos disponibles, incluyendo los de otros procesos. Un problema es que un proceso no puede controlar su propia frecuencia de fallos de página. Es el más utilizado, ya que aumenta el rendimiento del sistema.
- **Reemplazo Local.** Selecciona la victima solo de los marcos que le fueron asignados. Un problema es que un proceso se puede entorpecer al no poder aprovechar otras páginas que podrían tener muy poco uso.

9.7 Hiperpaginación (Trashing)

Un proceso que no tiene suficientes marcos para su cantidad de *páginas activas* (número de páginas que siempre esta usando), provocará un fallo de página a cada instante, por lo que comenzará a estar en un estado de frenética paginación, pasando más tiempo paginando que ejecutando, y esto se denomina *hiperpaginación.*

9.7.1 Causa de la Hiperpaginación

La hiperpaginación trae serios problemas de desempeño. El SO supervisa el aprovechamiento del CPU y cuando detecta que es bajo, introduce un nuevo proceso para aumentar la multiprogramación. En un sistema con reemplazo global, este proceso puede generar un fallo si tiene pocas páginas y al hacerlo le quita páginas a otros procesos, los otros procesos hacen lo mismo, y todos

comienzan a hiperpaginar, terminando con un aprovechamiento del CPU peor al que había antes. Al bajar el aprovechamiento, el SO introduce otro proceso, y así sucesivamente hasta desplomar el rendimiento del sistema.
Al agregar procesos, el rendimiento aumenta, hasta llegar a un punto máximo de procesos concurrentes. Si se siguen agregando procesos, el rendimiento cae abruptamente. A partir de ese punto, para aumentar el rendimiento, hay que disminuir la cantidad de procesos concurrentes.
Este efecto se puede limitar utilizando un algoritmo de reemplazo local (o por prioridad). Pero para evitar la hiperpaginación hay que darle a cada proceso los marcos que necesita.

9.7.2 Modelo de Conjunto de Trabajo

Se basa en el supuesto de localidad. Este modelo emplea un parámetro D para definir la *ventana del conjunto de trabajo*. De lo que se trata es de examinar las D referencias a páginas más recientes, y esas D páginas es el *conjunto de trabajo*. Si una página esta en uso activo estará en el conjunto de trabajo; si ya no se esta usando saldrá del conjunto, D unidades de tiempo después de su ultima referencia. Así, el conjunto de trabajo es una aproximación de localidad del programa.
El sistema operativo vigila el conjunto de trabajo de cada proceso y le asigna un número de marcos igual a su tamaño. Si hay marcos libres, puede iniciar otro proceso. Si la suma de los tamaños de los conjuntos de trabajo aumenta hasta exceder el numero total, el SO suspenderá un proceso (y se iniciará más tarde).
Esto evita la hiperpaginación y mantiene la multiprogramación lo más alta posible. El problema es que es difícil seguir la pista a los conjuntos de trabajos.

9.7.3 Frecuencia de Fallos de Pagina

Podemos establecer límites superiores e inferiores para la frecuencia de fallos de páginas deseada. Si la frecuencia de fallos real excede el límite superior, se le da al proceso otro marco y si baja del inferior se le quita uno.
Evitamos así la hiperpaginación, pero en el caso de que la frecuencia de fallos de página aumente y no haya marcos disponibles, deberemos seleccionar un proceso y suspenderlo.

9.8 Otras Consideraciones

9.8.1 Prepaginación

Es un intento por evitar el alto nivel de fallos de página al iniciar el proceso. La estrategia consiste en traer a memoria en una sola operación todas las páginas que se necesiten, para ello se debe guardar junto con cada proceso, una lista de las páginas de su *conjunto de trabajo*. Esto puede ser ventajoso en algunos casos. Hay que considerar si el costo de la prepaginación es menor que el de atender los fallos de página correspondientes.

9.8.2 Tamaño de página

La memoria se aprovecha mejor con páginas pequeñas, pero como cada proceso activo mantiene su tabla de páginas es preferible páginas grandes (para tener una tabla más pequeña).
Otro problema a considerar es el tiempo que toma leer o escribir una página, y este es un argumento a favor de las páginas grandes.
Con páginas pequeñas, tenemos mejor *definición*, solo traemos a memoria lo realmente necesario, con páginas más grandes, debemos transferir todo lo que contenga la página, sea necesario o no. Pero cuanto más grandes sean las páginas, menor es la cantidad de fallos de página (que producen gastos extra).
La tendencia histórica es a páginas más grandes, pero no hay una postura definida.

9.8.3 Tablas de Páginas Invertidas

El propósito es reducir la cantidad de memoria física necesaria para las traducciones de direcciones virtuales a físicas. El problema es que no contienen la información necesaria para trabajar con paginación por demanda, necesitando una tabla de

páginas externa para cada proceso, que contengan esa información.

9.8.4 Estructura del Programa

La paginación por demanda es transparente al programa de usuario. Sin embargo se puede mejorar el desempeño del sistema si se conoce.
Ver ejemplo página 325 del Silberschatz (útil para el práctico).

9.8.5 Interbloqueo de E/S

A veces es necesario permitir que se *bloqueen* o fijen algunas páginas en memoria. Esto implica que no se reemplacen páginas que están esperando una operación de E/S, o páginas que son utilizadas como buffers para el mismo fin (E/S). La forma de implementarlo es asociando un bit de bloqueo a cada marco.
Otro motivo de bloqueo, es cuando un proceso de baja prioridad, trae a memoria una página, y antes de usarla es desplazado por un proceso de mayor prioridad. El proceso de baja prioridad va a la cola de procesos listos, y permanece allí por bastante tiempo (debido a su baja prioridad). Los marcos de memoria van a ir siendo reemplazados, hasta que se reemplaza el marco del proceso de baja prioridad.
Estamos desperdiciando la labor invertida en traer la página del proceso de baja prioridad y para impedirlo, podemos bloquearla.
El bloqueo es peligroso, si el bit se enciende pero nunca se apaga (por algún fallo del sistema), ya que el marco queda inutilizado.

9.8.6 Procesamiento en Tiempo Real

La memoria virtual trae retardos a los procesos, por lo cual no es posible utilizarla en sistemas de tiempo real.

9.9 Segmentación por Demanda

La paginación por demanda se considera el sistema de memoria virtual más eficiente, pero se requiere de mucho hardware para implementarlo. Si falta el hardware una opción posible es la *segmentación por demanda*.
El funcionamiento es muy similar a la paginación por demanda, al igual que sus algoritmos y soluciones a problemas. Pero, el gasto es mayor ya que los segmento son de tamaño variable y esto complica los algoritmos.

10 Interfaz con el sistema de archivos

El sistema de archivos consiste en dos partes distintas: una colección de *archivos*, y una *estructura de directorios* que los organiza. Algunos sistemas de archivos tienen *particiones* que sirven para separar física o lógicamente grandes colecciones de directorios.

10.1 El concepto de archivo

El SO abstrae las propiedades físicas de los dispositivos (*no volátiles*) de almacenamiento para definir una unidad de almacenamiento lógica, *el archivo*, y establece una correspondencia entre ellos.
Es una colección de información relacionada que se graba en almacenamiento secundario y se le asigna un nombre. No es posible escribir datos, si no es dentro de un archivo.
El creador del archivo define su contenido, y su estructura (programa fuente, objeto, ejecutable, de texto, etc.).

10.1.1 Atributos de archivo

Los usados generalmente son:

- **Nombre:** Información que se mantiene en forma comprensible para los seres humanos.
- **Tipo:** Se necesita saber en sistemas que reconocen distintos tipos de archivo.
- **Ubicación:** Es un puntero a un dispositivo y a la posición del archivo en el dispositivo.
- **Tamaño:** Tamaño actual del archivo, y tal vez el tamaño máximo permitido.
- **Protección:** Determina quien puede leer, escribir, ejecutar, etc. el archivo.
- **Hora, fecha e identificación de usuario:** Pueden ser útiles para la protección, seguridad y control de uso.

Esta información se guarda en la estructura de directorios.

10.1.2 Operaciones con archivos

El SO cuenta con llamadas al sistema para trabajar con archivos. Algunas son:

- **Crear un archivo:** Es en dos pasos.
  - Encontrar espacio en el sistema de archivos
  - Insertar una entrada para el archivo en el directorio (registrando el nombre y ubicación en el sistema de archivos)
- **Escribir un archivo:** Especifica el nombre y la información. Con el nombre se efectúa una búsqueda para encontrarlo en el directorio, manteniendo un puntero de *escritura* indicando la posición donde se va a escribir.
- **Leer un archivo:** Se efectúa una búsqueda por nombre, y se mantiene un puntero de *lectura* al archivo. Generalmente se usa el mismo puntero para lectura y escritura, ahorrando espacio.
- **Reubicarse dentro de un archivo:** Se cambia la posición actual del archivo (el puntero).
- **Eliminar un archivo:** Lo buscamos en el directorio, liberamos el espacio que ocupa, y borramos la entrada del directorio.
- **Truncar un archivo:** Permite dejar inalterados todos los atributos de un archivo, y borra su contenido, haciendo que su longitud sea cero.

Hay otras operaciones como cambio de nombre, copia, etc.
Para no tener que buscar el archivo cada vez que voy a realizar una operación, algunos sistemas guardan la *tabla de archivos abiertos* donde colocan una entrada cuando el archivo se utiliza por primera vez. Cuando dejo de usarlo, se borra esa entrada.
La implementación de las operaciones abrir y cerrar en un entorno multiusuario es más complicada, ya que varios usuarios podrían abrir un archivo al mismo tiempo. Para implementar esto se utilizan dos tablas, una que contiene los archivos abiertos de cada proceso, y otra que contiene la información de su ubicación, punteros, etc. Los elementos de información son:

- **Puntero al archivo:** Es exclusivo para cada proceso que está trabajando con el archivo, por lo que debe mantenerse aparte de los atributos del archivo en disco.
- **Contador de aperturas del archivo:** Como varios procesos pueden abrir un archivo, el sistema debe esperar a que todos lo cierren antes de eliminar la entrada correspondiente. Este contador sigue la pista del número de aperturas y cierres, y llega a cero, después del último cierre.
- **Ubicación del archivo en disco:** Es la información necesaria para localizar el archivo en el disco, y se mantiene en memoria, para no tener que leerla del disco en cada operación.

10.1.3 Tipos de archivos

Si un sistema reconoce el tipo de un archivo, puede trabajar con él de manera razonable. Una forma para implementar el tipo, es incluirlo dentro del nombre del archivo. Se divide el nombre en dos partes (nombre y extensión), separados por un punto. Con solo ver la extensión el sistema reconoce el tipo, y sabe que operaciones puede realizar con él.
Otros sistemas guardan un atributo del creador del archivo, y al abrir el archivo, el sistema por medio de ese atributo, sabe con que programa debe abrirlo.

10.1.4 Estructura de los archivos

Los tipos pueden servir para determinar la estructura interna del archivo. Además, se podría obligar a que ciertos archivos

tengan una estructura determinada que el sistema entiende.
Por cada estructura de archivo distinta, el sistema debe definir el código que la maneja. Otra forma es que cada programa reconozca sus archivos, y tenga el código para trabajar con ellos, dejando al sistema la obligación de reconocer, por lo menos, la estructura del archivo ejecutable, para poder cargar los programas.
Si hay pocas estructuras, la programación se dificulta, y si hay demasiadas, el sistema crece excesivamente.

10.1.5 Estructura interna de los archivos

Localizar una posición en un archivo, puede ser difícil para el sistema operativo, puesto que los tamaños de los archivos nunca coinciden con los tamaños de los bloques del dispositivo de almacenamiento. Una solución común es *empaquetar* varios registros lógicos en bloques físicos, considerando un archivo como una secuencia de bloques.
Al asignar el espacio por bloques podemos crear fragmentación interna.

10.2 Métodos de acceso

Hay varias formas de acceder a la información del archivo.

10.2.1 Acceso secuencial

La información se procesa en orden, un registro tras otro. Es utilizado por los editores, compiladores, operaciones de lectura y escritura, etc.
Está basado en un modelo de archivo de cinta, y funciona bien, tanto en dispositivos de acceso secuencial como aleatorio.

10.2.2 Acceso directo

Está basado en el modelo de los discos, ya que permiten el acceso aleatorio a cualquier bloque del archivo, y son útiles para obtener acceso inmediato a grandes cantidades de información, como por ejemplo, las bases de datos.
Se utilizan números de bloque relativos, para poder colocar el archivo en cualquier parte del disco, comenzando por ejemplo en el bloque 15678 (bloque relativo 0). Es fácil simular el acceso secuencial en un archivo de acceso directo.

10.2.3 Otros métodos de acceso

Generalmente implican la construcción de un *índice* para el archivo, conteniendo punteros a los distintos bloques, permitiendo efectuar búsquedas en archivos grandes, sin realizar mucha E/S. Se utiliza en archivos donde debo buscar por una clave determinada.

10.3 Estructura de directorios

Para administrar los datos del sistema de archivos, hay que organizarlos. Primero se divide el sistema de archivos en *particiones*, donde cada disco contiene al menos una. Hay otros sistemas que permiten que las particiones sean mayores que un disco.
Cada partición tiene información acerca de los archivos que hay en ella, manteniéndola como entradas de una *tabla de contenido del volumen*.
Operaciones en un directorio:

- **Buscar un archivo:** Se puede buscar por nombre, o listar todos los archivos cuyos nombres coinciden con un patrón.
- **Crear un archivo:** Crea archivo y lo agrega al directorio.
- **Eliminar un archivo:** Elimina el archivo del directorio.
- **Listar un directorio:** Lista los archivos que se encuentran en él.
- **Cambiar el nombre de un archivo:** Cambio su nombre, y puedo modificar su posición dentro de la estructura de directorios.
- **Recorrer el sistema de archivos:** Puedo acceder a todos los directorios y archivos dentro de una estructura.

10.3.1 Directorio de un solo nivel

Todos los archivos se guardan en un mismo directorio, pero tiene limitaciones cuando el número de archivos aumenta, o hay más de un usuario, ya que deben tener nombres únicos.

10.3.2 Directorio de dos niveles

Se puede crear un directorio individual para cada usuario. Al iniciar la sesión, cada usuario accede solo a su directorio.
Si permito acceder a archivos de otros usuarios, para acceder al archivo, debo indicar, el *camino* (path) y el nombre del archivo. Por ejemplo, si quiero acceder a mi archivo *prueba* simplemente lo nombro, pero si quiero acceder al del usuario b, debo poner */usuariob/prueba*.

10.3.3 Directorios con estructura de árbol

Esta generalización permite a los usuarios crear sus propios subdirectorios y organizar sus archivos de manera acorde. Cada directorio, contiene un conjunto de archivos o subdirectorios.
Si el usuario quiere un archivo del directorio actual, simplemente accede a él, si quiero un archivo de otro directorio, debo especificar el path. Hay llamadas al sistema que permiten moverse dentro de los directorios.
Los path pueden ser:

- **Absolutos:** Parte de la raíz, y sigue el camino hasta el archivo.
- **Relativos:** Define un camino a partir del directorio actual.

Una decisión de política es la eliminación de directorios. Si está vacío, se puede borrar fácilmente. Si contiene archivos o subdirectorios, se debe decidir:

- **No lo elimino:** Obligo al usuario que lo vacíe antes de borrarlo, pero podría ser un trabajo engorroso si contiene subdirectorios.
- **Lo elimino:** El sistema se encarga de eliminar todos los archivos y subdirectorios de ese directorio recursivamente.

10.3.4 Directorios de grafo acíclico

Hay subdirectorios que se comparten, existe en el sistema de archivos, en dos o más lugares simultáneamente. No hay dos (o más) copias del archivo, sino que puedo acceder al mismo archivo desde dos lugares diferentes. Cualquier modificación al archivo, se refleja en todos los usuarios que lo están usando.
Hay varios problemas, ya que un mismo archivo puede tener múltiples nombres de camino absoluto, y además, cuando se elimina algún archivo, podrían quedar “punteros colgantes”.
Una forma de manejarlo es teniendo un contador de enlaces, y eliminando el archivo solamente cuando ya no hay más enlaces a él.

10.3.5 Directorio de grafo general

Un problema del caso anterior es asegurar que no haya ciclos. En este caso, se deben diseñar cuidadosamente los algoritmos de búsqueda, para que no se den ciclos infinitos. Una solución es limitando el número de directorios a los que se accede en la búsqueda.
También surgen problemas al eliminar archivos, ya que pueden no quedar referencias al archivo, y tener un contador de referencias distinto de cero. Para ello se utiliza un esquema de *recolección de basura* (garbage collector) que libera esos espacios de disco, pero es muy costoso.

10.4 Protección

Se debe proteger la información tanto de daños físicos (problemas de hardware, corte del suministro de energía, etc.) como de accesos indebidos.

10.4.1 Tipos de acceso

En los sistemas que no permiten el acceso a archivos de otros usuarios no se necesita protección. En los demás sistemas se necesita un *acceso controlado,* controlando alguna de las operaciones relacionadas a archivos como: leer, escribir, ejecutar, eliminar, listar, etc.
La protección también depende del tamaño del sistema, y de la importancia del contenido de los archivos.

10.4.2 Listas y grupos de acceso

Se puede hacer que el acceso dependa de la identidad del usuario, asociando a cada archivo y directorio una *lista de acceso* que especifique las operaciones permitidas para cada usuario. El principal problema es la longitud de estas listas, ya que si queremos que todos los usuarios lean un archivo, debemos listarlos a todos. Hay dos consecuencias:

- La construcción de la lista podría ser tediosa e inútil cuando no se conocen por anticipado la lista de usuarios.
- La entrada del directorio debe ser variable, complicando la administración del espacio.

10.4.3 Otras estrategias de protección

Se puede asociar una contraseña a cada archivo pero tiene desventajas como:

- Cada usuario debe recordar un gran número de contraseñas.
- Si se usa una sola contraseña para todos los archivos, el sistema estaría en peligro si alguien la descubre.

También es necesario proteger los directorios, complicándose en los sistemas en que los archivos pueden tener varios nombres de camino (grafos), ya que se debe limitar el acceso dependiendo del path.

10.4.4 Un ejemplo: UNIX

Cada subdirectorio (al igual que los archivos) tiene asociados tres campos (propietario, grupo y universo) cada uno con los tres bits rwx (lectura, escritura y ejecución).

10.5 Semántica de consistencia

Indica que debe hacer el sistema cuando un usuario modifica un archivo compartido (que deben ver los demás usuarios que están accediendo a ese archivo).

10.5.1 Semántica de UNIX

- Las escrituras de un archivo abierto hechas por un usuario son visibles inmediatamente a todos los usuarios que lo tienen abierto.
- Cuando los usuarios comparten el puntero a la posición actual dentro del archivo, el adelanto del puntero por parte de uno, afecta a todos los demás.

Un archivo está asociado a una única imagen física.

10.5.2 Semántica de sesión

- Las escrituras de un usuario no son visibles para los otros usuarios que tienen abierto el mismo archivo simultáneamente.
- Una vez cerrado el archivo, solo los usuarios que lo abran posteriormente verán los cambios efectuados. Los ejemplares ya abiertos, no los reflejan.

Un archivo se puede asociar temporalmente a varias imágenes al mismo tiempo.

11 Implementación del sistema de archivos

El sistema de archivos proporciona el mecanismo para almacenar en línea y acceder a los datos y programas. Reside de manera permanente en almacenamiento secundario, conteniendo una gran cantidad de datos.

11.1 Estructura del sistema de archivos

Las transferencias entre la memoria y el disco se efectúan en unidades de *bloques*, mejorando de esta manera la eficiencia de E/S. Cada bloque ocupa uno o más *sectores*. El tamaño de los sectores, que varía entre 32 y 4096 bytes, por lo regular es de 512 bytes.
Dos características importantes de los discos (que los convierte en el medio de almacenamiento secundario más común) son:

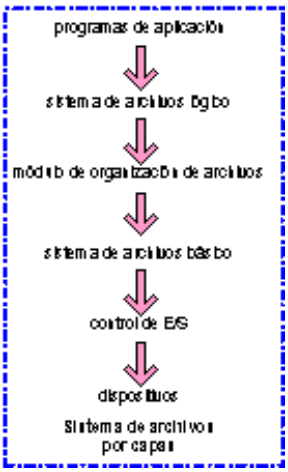
1. **Se pueden rescribir en el mismo lugar:** Es posible leer un bloque del disco, modificar el bloque y volverlo a escribir en el mismo lugar.
2. **Podemos acceder directamente a cualquier bloque de información del disco:** El cambio de una archivo a otro sólo requiere mover las cabezas de lectura-escritura y esperar que el disco gire.

11.1.1 Organización del sistema de archivos

Hay dos problemas de diseño del sistema de archivos:

- Definir qué aspecto debe presentar el sistema a los usuarios. Implica la definición de un archivo y sus atributos, las operaciones y la estructura de directorios para organizarlos.
- Crear algoritmos y estructuras de datos para establecer correspondencias entre el sistema de archivos lógico y los dispositivos de almacenamiento secundario.

Cada nivel del diseño aprovecha las funciones de los niveles inferiores para crear nuevas funciones que se usarán en los niveles superiores.



- **Control de E/S:** Consta de *drivers* o *controladores de dispositivos* y manejadores de interrupciones para transferir información entre la memoria y el sistema de disco. Se puede considerar a un *driver* como un traductor, cuya entrada consiste en instrucciones de alto nivel y su salida en instrucciones de bajo nivel específicas para el hardware.
- **Sistema de archivos básico:** Sólo necesita emitir órdenes genéricas al driver de dispositivo apropiado para leer y escribir bloques físicos en el disco. Cada bloque físico se identifica con su dirección numérica en disco (unidad 1, cilindro 73, sector 10).
- **Módulo de organización de archivos:** Conoce los archivos, sus bloques lógicos y sus bloques físicos, trabajando como traductor de direcciones de bloques (lógicas a físicas) para que el sistema de archivos básico realice la transferencia. Los bloques lógicos de cada archivo de numeran del 0 a N. También incluye el administrador de espacio libre.
- **Sistema de archivos lógico:** Emplea la estructura de directorios para proporcionar al módulo de organización la información que éste necesita, a partir de un nombre de archivo. También se encarga de la protección y la seguridad.

Para crear un archivo nuevo, un programa de aplicación llama al sistema de archivos lógico, éste trae el directorio apropiado a la memoria, lo actualiza con la nueva entrada, y lo vuelve a escribir en el disco.

Algunos sistemas operativos, como UNIX, tratan los directorios como si fueran archivos, con un campo de tipo. Otros, como Windows NT, implementan llamadas al sistema distintas para los archivos y para los directorios.
Antes de que el archivo pueda ser utilizado para una operación de E/S, es precioso abrirlo. Cuando un archivo se abre, se busca la entrada deseada en la estructura de directorios. La información correspondiente, como su tamaño, propietario, permisos de acceso y ubicación de los bloques de datos, se copia en una tabla en la memoria llamada *tabla de archivos abiertos*. El índice que corresponde al archivo abierto en esta tabla se devuelve al programa de usuario, y todas las referencias posteriores de hacen a través del índice.
En tanto no se cierre el archivo todas las operaciones con él se harán a través de la tabla de archivos abiertos. Cuando todos los usuarios que abrieron un archivo lo cierran, la información actualizada del archivo se copiará de vuelta en la estructura de directorios en el disco.

11.1.2 Montaje de sistemas de archivos

Un sistema de archivos debe *montarse* antes de estar disponible para los procesos del sistema.
El procedimiento es el siguiente:

1. Se proporciona al sistema operativo el nombre del dispositivo, junto con la posición dentro de la estructura de archivos en la que debe adosar el sistema de archivos (punto de montaje).
2. El sistema operativo verifica que el dispositivo contenga un sistema de archivos válido; esto lo hace pidiendo al driver del dispositivo que lea el directorio del dispositivo y compruebe que tenga el formato esperado.
3. El sistema operativo toma nota en su estructura de directorios de que un sistema de archivos está montado en el punto de montaje especificado.

11.2 Métodos de asignación

El problema principal es como distribuir los archivos en el disco de forma de aprovechar el espacio, y poder acceder rápidamente a ellos.

11.2.1 Asignación contigua

Requiere que cada archivo ocupe un conjunto de bloques contiguos en el disco. Las direcciones de disco definen un ordenamiento lineal en él, y el número de búsquedas de disco necesarias para acceder a archivos asignados contiguamente es mínimo. Si el archivo tiene n bloques, y comienza en la posición *b*, ocupará los bloques *b*, *b + 1*, *b + 2*, ... , *b + n - 1*.
La asignación contigua permite manejar acceso tanto secuencial como directo. Pero la dificultad principal es encontrar espacio para un archivo nuevo, y se deben contar con los algoritmos de *asignación dinámica de almacenamiento* (como satisfacer una solicitud de tamaño n a partir de la lista de huecos libres). Y se utilizan comúnmente las estrategias de primer ajuste y mejor ajuste.
Estos algoritmos padecen el problema de la *fragmentación externa* y a medida que archivos se asignan y borran, el espacio libre en disco se divide en trozos pequeños. Para ello se usa la *compactación* pero insume gran cantidad de tiempo (especialmente en discos grandes), y se debería hacer muy seguido.
Otro problema importante es determinar cuánto espacio se necesita para un archivo, ya que al crearlo, debo especificar el tamaño que va a tener en el futuro, para que el sistema busque un lugar donde ubicarlo. Además, si trabajamos con archivos que crecen lentamente durante un largo período de tiempo, debemos asignarle un gran tamaño al crearlo, el cual va a estar la mayor parte del tiempo desocupado, creando *fragmentación interna.*
Algunos sistemas operativos utilizan un esquema de asignación contigua modificado, en el que inicialmente se asigna un trozo contiguo de espacio y luego, cuando ese espacio deja de ser suficiente, se añade otro trozo de espacio contiguo, una *extensión*, a la asignación inicial.

11.2.2 Asignación enlazada

La asignación enlazada resuelve todos los problemas de la asignación contigua. Cada archivo es una lista enlazada de bloques de disco, los cuales pueden estar dispersos en cualquier parte del disco. El directorio contiene un puntero al primer y al último bloque del archivo. Cada bloque contiene un puntero al siguiente bloque.

Para crear un archivo nuevo, simplemente creamos una nueva entrada en el directorio. El cual recibe el valor nill para indicar un archivo vacío, y también se asigna cero al campo de tamaño. Para leer un archivo, basta con leer los bloques siguiendo los punteros de un bloque al siguiente, y para escribir, se escribe en un bloque libre, y se lo enlaza al final del archivo.
No hay fragmentación externa y no hay necesidad de declarar el tamaño de un archivo en el momento de crearlo.
El principal problema es que sólo puede usarse efectivamente para archivos de acceso secuencial ya que para acceder directamente a un bloque necesito ir siguiendo los punteros desde el primer bloque, hasta el que busco, teniendo un acceso a disco para cada puntero que leo, y a veces una búsqueda de disco.
Otra desventaja de la asignación enlazada es el espacio que ocupan los punteros. Cada archivo requiere un poco más de espacio del que requeriría en otro caso. La solución usual a este problema es juntar bloques en múltiplos, llamados *clusters*, y asignar los clusters en lugar de los bloques. Pero trae el problema de *fragmentación interna*.
Un problema adicional surge cuando un puntero se pierde o daña, por lo tanto, debería guardar información adicional para comprobar la correctitud, generando más gastos.
Una variación importante del método de asignación enlazada es usar una tabla de asignación de archivos (FAT, file-allocation table). Se aparta una sección del disco al principio de cada partición para guardar en ella la tabla, la cual tiene una entrada para cada bloque del disco y está indizada por número de bloque. Cada entrada, tiene el índice del siguiente bloque del archivo, continuando hasta el último bloque, para el cual la entrada es un valor especial llamada EOF. Los bloques desocupados se indican con un valor de cero en la tabla. Para asignar un bloque nuevo a un archivo, basta con encontrar la primera entrada de la tabla que valga cero y sustituir el valor EOF anterior por la dirección del nuevo bloque.
El esquema de asignación FAT puede dar pie a un número significativo de movimientos de la cabeza del disco, a menos que la FAT se mantenga en caché.

11.2.3 Asignación indizada

Si no se usa una FAT, la asignación enlazada no puede apoyar un acceso directo eficiente, ya que los punteros a los bloques están dispersos por todo el disco (junto con los bloques). La asignación indizada resuelve este problema al reunir todos los punteros en un mismo lugar: el *bloque índice*.
Cada archivo tiene su propio bloque índice, que es una matriz de direcciones de bloques de disco. La i-ésima entrada del bloque índice apunta al i-ésimo bloque del archivo.
Cuando se crea el archivo, se asigna nil a todos los apuntadores del bloque índice. La primera vez que se escribe el i-ésimo bloque, se obtiene un bloque del administrador de espacio libre y su dirección se coloca en la i-ésima entrada del bloque índice.
No sufre fragmentación externa, pero sí desperdicia espacio ya que el gasto extra de los punteros del bloque índice generalmente es mayor que el de los punteros de la asignación enlazada.
Otro problema es el tamaño del bloque índice, ya que si es demasiado pequeño no podrá contener suficientes punteros para un archivo grande. Los mecanismos para resolver este problema son:

- **Esquema enlazado:** Para manejar archivos grandes, podríamos enlazar varios bloques índice (ya que un bloque índice ocupa normalmente un bloque de disco).
- **Índice multinivel:** Usar un bloque índice de primer nivel que apunte a un conjuntos de bloques índice de segundo nivel, que a su vez apuntan a los bloques de disco.
- **Esquema combinado (i-nodo de Linux):** Guardar los primeros, (por ejemplo) 15 punteros del bloque índice en el bloque índice del archivo.
    - Los primeros 12 punteros apuntan a *bloques directos* (bloques con datos del archivo).
    - Los siguientes tres apuntan a *bloques indirectos*.
        - El primer puntero de bloque indirecto es la dirección de un *bloque indirecto simple*: un bloque índice que no contiene datos, sino las direcciones de bloques que sí contienen datos.
        - Luego viene un puntero a un *bloque indirecto doble*, que contiene la dirección de un bloque que contiene las direcciones de bloques que contienen punteros a los bloques de datos reales.
        - El último puntero contendría la dirección de un *bloque indirecto triple*. Un puntero de archivo de 32 bits sólo, alcanza para 4 gigabytes.

Los bloques índice se pueden colocar en un caché, pero los bloques de datos podrían estar dispersos por toda una partición, esto

causaría problemas de desempeño.

11.2.4 Desempeño

La asignación contigua sólo requiere un acceso para obtener un bloque de disco, y es simple acceder al i-ésimo bloque. La asignación enlazada es buena para acceso secuencial, pero no debe usarse con una aplicación que requiere acceso directo.
Algunos sistemas manejan los archivos de acceso directo usando asignación contigua, y los de acceso secuencial, con asignación enlazada, por lo tanto el tipo de acceso que se efectuará se declara en el momento de crear el archivo. Un archivo creado para acceso secuencial se enlazará y no podrá usarse para acceso directo. Un archivo creado para acceso directo será contiguo y podrá apoyar el acceso tanto directo como secuencial, pero su longitud máxima deberá declararse en el momento en que se cree. De todas formas, los archivos pueden convertirse de un tipo a otro, creando un nuevo archivo y copiando los datos.
La asignación indizada es más compleja. Si el bloque índice ya está en la memoria, el acceso podrá ser directo, pero para mantenerlo necesito un espacio considerable. El desempeño de la asignación indizada depende de la estructura de índices, del tamaño del archivo y de la posición del bloque deseado.
Algunos sistemas combinan la asignación contigua con la indizada usando asignación contigua si el archivo es pequeño y cambiando automáticamente a una asignación indizada si el tamaño del archivo aumenta.

11.3 Administración del espacio libre

Para seguir la pista al espacio libre en el disco, el sistema mantiene una *lista de espacio libre* en la que se registran todos los bloques de disco que están libres. Para crear un archivo, buscamos en la lista de espacio libre la cantidad de espacio necesaria y la asignamos al nuevo archivo. El espacio se elimina de la lista. Hay varias formas de implementarla.

11.3.1 Vector de bits

Cada bloque se representa con un bit. Si el bloque está libre, el bit es 1; si el bloque está asignado, el bit es 0.
Resulta relativamente sencillo y eficiente encontrar el primer bloque libre, o n bloques libres consecutivos en el disco, pero son ineficientes si el vector entero no se mantiene en la memoria principal, y esto solo es posible si el disco no es demasiado grande.

11.3.2 Lista enlazada

Se enlazan entre sí los bloques de disco libres, manteniendo un puntero al primer bloque libre en una posición especial del disco y colocándolo en caché en la memoria.
Este esquema no es eficiente ya que para recorrer la lista necesitaríamos leer cada bloque, ocupando una cantidad sustancial de tiempo de E/S, aunque no se necesita hacerlo frecuentemente.
El método FAT incorpora la contabilización de bloques libres en la estructura de datos de asignación, así que no necesita otro método aparte.

11.3.3 Agrupamiento

Almacena las direcciones de n bloques libres en el primer bloque libre permitiendo encontrar rápidamente las direcciones de un gran número de bloques libres.

11.3.4 Conteo

Aprovecho el hecho de que generalmente, al liberar un bloque, también libero los contiguos. En lugar de mantener una lista de direcciones de bloques libres, mantenemos la dirección del primer bloque libre y el número de n bloques libres contiguos que siguen al primero.

11.4 Implementación de directorios

11.4.1 Lista lineal

El método más sencillo para implementar un directorio es usar una lista lineal de nombres de archivo con punteros a los bloques de datos, con la desventaja de que requiere una búsqueda lineal para hallar una entrada específica, consumiendo mucho tiempo al ejecutarse (además, la búsqueda es algo que se requiere muy seguido).
Una lista ordenada permite realizar una búsqueda binaria y reduce el tiempo promedio de las búsquedas, pero dificulta la creación y eliminación de archivos. Una ventaja de la lista ordenada es que se puede producir un listado de directorio en orden sin pasar por un paso de ordenamiento aparte.

11.4.2 Tabla de dispersión (hash table)

Las entradas de directorio se guardan en una lista lineal, pero también se usa una estructura de datos de dispersión. La tabla de dispersión recibe un valor que se calcula a partir del nombre de archivo y devuelve un puntero a la entrada de ese archivo en la lista lineal. Esto puede reducir mucho el tiempo de búsqueda en directorios, y la inserción y eliminación también son más o menos directas.
Los principales problemas de la tabla de dispersión son que su tamaño generalmente es fijo y que la función de dispersión depende del tamaño de dicha tabla. Además se deben proveer mecanismos para el manejo de colisiones.

11.5 Eficiencia y desempeño

11.5.1 Eficiencia

El uso eficiente del espacio en disco depende mucho de los algoritmos de asignación de disco y manejo de directorios que se usen.
Otra cosa que debemos considerar son los tipos de datos que se mantienen en la entrada de directorio, ya que en muchos casos, la lectura implica que se deban escribir datos en esta estructura, trayendo el bloque a memoria, modificándolo y volviéndolo a guardar en disco.
También debemos tener en cuenta el tamaño de los punteros a utilizar, ya que si es pequeño (16 o 32 bits) limitamos el tamaño del archivo, y si son grandes (64 bits), ocupan demasiado espacio.

11.5.2 Desempeño

La mayor parte de los controladores de disco incluyen memoria local para formar un caché interno con el tamaño suficiente para almacenar pistas enteras a la vez, pudiendo pasarlas todos los datos a memoria a la vez (bajo el supuesto de que se van a volver a utilizar). Además se utilizan diferentes algoritmos de reemplazo del caché (LRU, etc.), dependiendo del tipo de acceso del archivo (secuencial, directo, etc). Por ejemplo para acceso secuencial puedo utilizar *lectura adelantada* copiando el bloque necesario y los subsecuentes, para ahorrar tiempo.
Otro método para mejorar el desempeño es apartar una sección de la memoria principal y tratarla como un disco virtual, o disco RAM. La diferencia entre un disco RAM y un caché de disco es que el contenido del disco RAM está totalmente bajo el control del usuario, mientras que el del caché de disco está bajo el control del sistema operativo. El inconveniente es que solo es útil para el almacenamiento temporal, ya que pierde los datos en casos de cortes de energía.

11.6 Recuperación

Puesto que los archivos y directorios se mantienen tanto en la memoria principal como en disco, se debe cuidar que un fallo del sistema no cause una pérdida de datos ni inconsistencia de los datos.

11.6.1 Verificación de consistencia

Si ocurre una caída del computador, la tabla de archivos generalmente se pierde (ya que está en memoria), y con ella cualesquier cambios recientes a los directorios de los archivos abiertos. Este suceso puede dejar al sistema de archivos en un

estado inconsistente, por lo tanto se debe utilizar un programa especial al rearrancar el sistema.
El *verificador de consistencia* compara los datos de la estructura de directorios con los bloques de datos en disco y trata de corregir las inconsistencias que encuentra, pero el éxito depende en gran parte de la estructura de archivos utilizada.

11.6.2 Respaldo y restauración

Se pueden usar programas del sistema para respaldar datos del disco en otro dispositivo de almacenamiento. La recuperación después de la pérdida de un archivo individual, o de todo un disco, es cuestión de *restaurar* estos datos desde el respaldo. Tipos de respaldo:

- **Total:** Copia todos los archivos del disco.
- **Incremental:** Copia todos los archivos que cambiaron luego del respaldo total.

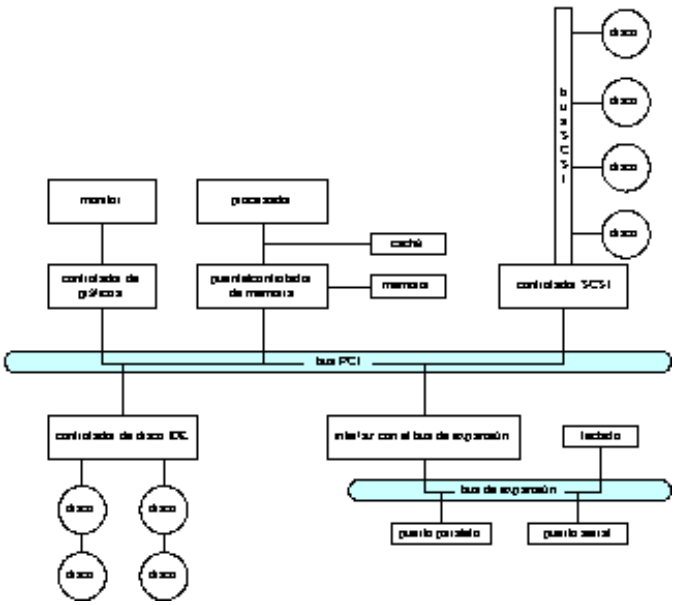
12 Sistemas de E/S

12.1 Generalidades

Dada la amplia variación en la función y velocidad de los dispositivos de entrada y salida, se requieren diversos métodos para controlarlos, estos constituyen el *subsistema* de E/S del núcleo (aísla el resto del núcleo de la complejidad de administrar los dispositivos de E/S).
Existe una tendencia a incorporar dispositivos mejorados a computadores y sistemas operativos ya existentes, por otro lado cada vez hay una variedad más amplia de dispositivos de E/S, siendo el reto incorporarlo a nuestros computadores y sistemas operativos. Para esto el núcleo de un sistema operativo se estructura de manera de usar módulos *controladores de dispositivos* o *drivers*, los cuales encapsulan los detalles y peculiaridades de los diferentes dispositivos, presentado una interfaz de acceso uniforme a estos.

12.2 Hardware de E/S

Un dispositivo se comunica con un sistema de computación enviando señales a través de un punto de conexión denominado *puerto* (como un puerto serial). Si uno o mas dispositivos usan un conjunto común de “cables” la conexión se llama *bus*. Mas formalmente un bus es conjunto de “cables” con un protocolo rígidamente definido que especifica un conjunto de mensajes que se pueden enviar por ellos.
Un *controlador* es una colección de circuitos electrónicos que pueden operar un *puerto* un *bus* o un *dispositivo*. Hay distintas complejidades de controladores por ejemplo:



- **Simple:** El controlador del puerto serial es un chip que viene usualmente incluido en la PC
- **Complejo:** El controlador de bus SCSI que debido a la complejidad del protocolo, se implementa con una tarjeta de circuito aparte que contiene memoria propia.

Otros dispositivos traen integrados sus propios controladores (disco IDE…).

El procesador se comunica con un controlador para llevar a cabo una transferencia de E/S a través de uno o más registros para señales de datos y de control. Esta comunicación se puede realizar con dos técnicas diferentes:

- Utilizar instrucciones de E/S especiales que especifican la transfieren bytes a través del bus, hasta los registros del

dispositivo apropiado.
- El controlador de dispositivo maneje la E/S como mapa de memoria (ej: controlador de gráficos), habiendo una correspondencia entre los registros de control de dispositivo y el espacio de direcciones del procesador. Esta técnica es más rápida ya que la escritura en la memoria, es más rápida que la emisión de instrucciones de E/S. Pero tiene como desventaja la posibilidad que falle algún programa, escribiendo en una región de memoria no deseada. Este problema se reduce, protegiendo la memoria.

Un puerto de E/S suele consistir en cuatro registros:

- **Registro de estado**: Contiene bits que el anfitrión puede leer, por ejemplo indican si ya se ejecuto la orden actual, si hay un byte disponible para leerse en el registro de entrada de datos, etc.
- **Registro de control:** El anfitrión puede escribir para iniciar una orden o cambiar el modo de un dispositivo.
- **Registro de entrada de datos:** Es leído por el anfitrión para obtener la entrada.
- **Registro de salida de datos:** Es escrito por el anfitrión para enviar salidas.

Los registros de datos suelen tener entre 1 y 4 bytes. Algunos controladores tienen chips FIFO que pueden contener varios bytes de datos de entrada o de salida con el fin de expandir la capacidad del controlador más allá del tamaño del registro de datos.

12.2.1 Escrutinio (Polling)

Para comunicarse, el anfitrión espera hasta que el dispositivo se desocupe. Cuando lo hace, el anfitrión pone los datos necesarios para que el dispositivo procese la entrada y salida, y cuando el controlador percibe que se han puestos datos, procesa la E/S pedida por el anfitrión.
En este modelo el anfitrión esta en espera activa o escrutinio. Si el controlador y el dispositivo son rápidos este método es razonable, pero si la espera es larga lo mejor es que el anfitrión conmute a otra tarea. Es mas eficiente que el controlador del hardware notifique a la CPU cuando el dispositivo este listo para recibir servicio, en vez de exigirle a la CPU que escrute repetidamente para detectar una finalización E/S. Este mecanismo se denomina *interrupción*.

12.2.2 Interrupciones

El hardware del CPU tiene una línea de *solicitud de interrupción* (IRQ), que la CPU detecta después de ejecutar cada instrucción. Si la CPU detecta que algún controlador ha generado una interrupción (a través dicha línea), esta guarda su estado actual (puntero a instrucción actual), y despacha la rutina de *manejador de interrupción* que se encuentra en una dirección fija de la memoria. El manejador determina la causa de la interrupción, realiza el procesamiento necesario y ejecuta una instrucción de *retorno de interrupción* (el CPU vuelve al estado previo a la interrupción).

Este mecanismo permite a la CPU responder a un suceso asincrónico. En SO más modernos se requiere un manejo de interrupciones más avanzado, el cual se implementa en la CPU y el hardware *controlador de interrupciones*, agregando al modelo actual funciones para:

- Capacidad de postergar interrupciones durante un proceso crítico.
- Despachar eficientemente al manejador de interrupciones, sin tener que recorrer todos los dispositivos para ver quien genero la interrupción.
- Tener interrupciones de múltiples niveles, para poder distinguir interrupciones de alta y baja prioridad y responder con el grado de urgencia apropiado.

La mayoría de las CPU tiene dos líneas de solicitud de interrupción:

- **Interrupción no enmascarable:** Reservada para sucesos como errores de memoria no recuperables.
- **Interrupción enmascarable:** Utilizadas por los controladores de dispositivos. Estas pueden ser desactivadas antes de ejecutar instrucciones críticas que no deben interrumpirse.

A cada interrupción se le asocia un número que en la mayor parte de las arquitecturas representa un desplazamiento dentro del *vector de interrupciones* que contiene las direcciones de memoria de los manejadores de interrupción.
Usualmente en computadores modernos hay más dispositivos que lugares en el vector, por lo tanto se utiliza la técnica de encadenamiento de interrupciones, en la cual cada elemento del vector apunta a una lista de manejadores de interrupción. Cuando se genera una interrupción, se invocan uno a uno los manejadores de la lista correspondiente, hasta que el apropiado atiende la solicitud. Esta estructura no genera un gasto tan alto como tener un vector de interrupciones enorme, ni es tan eficiente como despachar a un solo manejador de interrupciones.

Usos habituales de las interrupciones:

- **Manejo de excepciones:** Por ejemplo, división entre cero, acceder a memoria protegida o inexistente, intentar ejecutar una instrucción privilegiada en modo usuario, etc.
- **Manejo de memoria virtual:** Se producen interrupciones por cada fallo de página.
- **Llamadas al sistema:** Se solicitan servicios del núcleo, verificando los argumentos proporcionados por la aplicación, construyendo una estructura de datos para comunicarlos al núcleo, y luego ejecutando una *interrupción de software* o *trampa*. El hardware de interrupciones guarda el estado del código de usuario, conmuta al modo supervisor y despacha a la rutina del núcleo que implementa el servicio solicitado.

12.2.3 Acceso directo a la memoria (DMA)

Para aliviar a al CPU de estar continuamente atendiendo una larga transferencia de datos, se traspasa parte de este trabajo a un procesador de propósito especial llamado *controlador de acceso directo a la memoria* (DMA). Un controlador de DMA sencillo es un componente estándar de los PC, y las tarjetas de E/S generalmente contienen su propio hardware de DMA de alta velocidad para la comunicación con el bus.
Para realizar una transferencia DMA, el anfitrión escribe un bloque de órdenes de DMA en memoria, el cual contiene un puntero al origen de una transferencia, otro al final, y un contador del número de bytes a transferir. La CPU le indica al controlador DMA la dirección de ese bloque, y prosigue con sus labores. De esta forma, el controlador opera con el bus de memoria directamente, transfiriendo los bytes indicados sin la ayuda del CPU. Este apoderamiento del bus, impide momentáneamente el acceso a memoria de la CPU (aunque si puede acceder a los cachés), haciendo más lentas sus operaciones, aunque generalmente el DMA mejora el desempeño del sistema.
Algunas Arquitecturas pueden realizar transferencias con acceso directo a memoria virtual (DVMA) empleando traducción de memorias virtuales a físicas, por lo que lo hace más eficiente debido a que no utiliza la memoria principal.
Este acceso directo, mejora el desempeño, pero perjudica la seguridad y estabilidad del sistema, ya que los procesos pueden acceder a los controladores de dispositivos directamente.

12.3 Interfaz de E/S de las aplicaciones

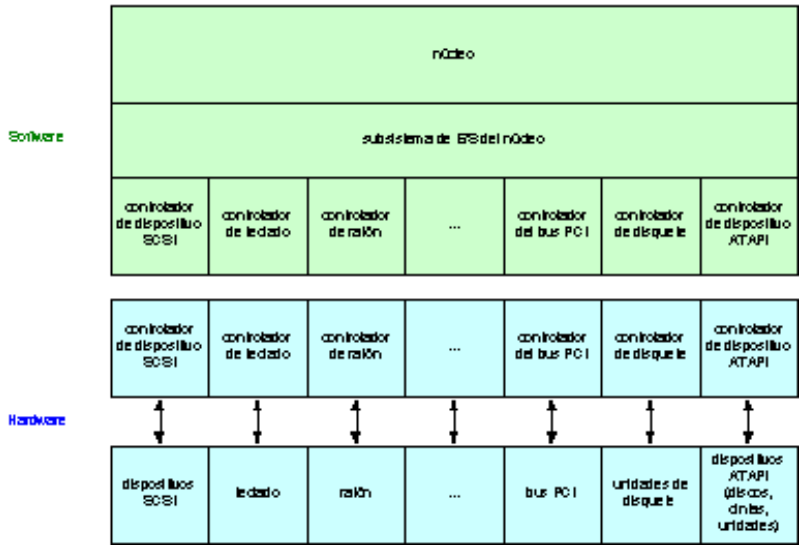
Se puede abstraer las diferencias detalladas de los dispositivos de E/S, para tratarlos de una forma uniforme y estandarizada, identificando unos cuantos grupos generales y accediendo a cada uno de estos grupos a través de unas cuantas operaciones estandarizadas: una **interfaz**.
Las diferencias reales se encapsulan en módulos del núcleo llamados *controladores de dispositivos* que internamente se adapta a cada dispositivo, pero que exportan una interfaz estándar. Esto simplifica la tarea del creador del SO y beneficia a los fabricantes de hardware, pero cada SO tiene su propia interfaz de drivers, por lo que un mismo dispositivo puede venderse con un driver distinto para cada plataforma (Windows, Linux, Solaris…).

Hay varios tipos de dispositivos de E/S:

- **Flujo de caracteres o Bloques:** Transfiere bytes ó bloques uno por uno, por ejemplo una terminal o un disco respectivamente.
- **Acceso secuencial o aleatorio:** Un dispositivo secuencial transfiere datos en un orden fijo y determinado (MODEM), mientras que el otro puede pedirle al dispositivo que se coloque en cualquiera de las posiciones de almacenamiento de

datos disponibles (cd-rom).

- **Sincrónico o asincrónico:** Un dispositivo sincrónico realiza transferencias de datos con tiempo de respuestas predecibles, mientras que en el otro estos tiempos son irregulares.
- **Compartible o dedicado:** Un dispositivo compartible puede ser utilizado en forma concurrente por varios procesos o hilo, no siendo así para un dispositivo dedicado.
- **Velocidad de operación:** Van de unos cuantos bytes/s a unos cuantos Gigabytes/s.
- **Lectura y Escritura:** Puede ser, lectura/escritura, solo lectura o solo escritura.



El acceso de las aplicaciones a los dispositivos se realiza usualmente a través de llamadas al sistema, que agrupan a los dispositivos en categorías.
Aunque las llamadas al sistemas exactas difieren de un SO a otro, las categorías de dispositivos son mas o menos estándares.

12.3.1 Dispositivo por bloques y por caracteres

La interfaz de dispositivos por bloques captura todos los aspectos necesarios para acceder a unidades de disco y otros dispositivos orientados a bloque. Las llamadas al sistema básicas permiten a una aplicación **leer** y **escribir**, y si es un dispositivo de acceso aleatorio también **buscar**, permitiendo que las aplicaciones puedan ignorar las diferencias de bajo nivel entre los dispositivos, accediendo a ellos a través de una interfaz de sistema de archivos.
El acceso a archivos con *mapa de memoria* en lugar de ofrecer operaciones de lectura y escritura, ofrece una interfaz que proporciona acceso al almacenamiento en disco a través de un arreglo de bytes en la memoria principal.
Las llamadas al sistema básicas de una interfaz de *flujo de caracteres* permiten a una aplicación **obtener** o **colocar** un carácter. Se utiliza para dispositivos como teclados, ratones, MODEM, etc.

12.3.2 Dispositivos de Red

La mayor parte de los SO proporcionan una interfaz de E/S de red diferente a la utilizada para discos. La interfaz que utilizan muchos SO (Unix, WinNT) es la interfaz de socket de red.
Las llamadas al sistema de la interfaz socket permiten a una aplicación crear un socket, conectar un socket local a una dirección remota, detectar si una aplicación se “enchufa” en el socket local y enviar y recibir paquetes por la conexión.
Esta interfaz también ofrece una función llamada seleccionar que administra un conjunto de sockets. Una llamada a esta función devuelve información acerca de cuales sockets tiene un paquete que espera ser recibido y cuales tienen espacio para aceptar un paquete que se enviara. El empleo de esta función elimina la espera activa, que de otra forma serían necesarias para la E/S de red, y facilitan la creación de aplicaciones distribuidas que puede usar cualquier hardware de red.

12.3.3 Relojes y temporizadores

Las computadoras cuentan con ello para:

- Dar la hora actual
- Dar el tiempo transcurrido
- Establecer un temporizador para iniciar la operación X en el instante T.

Son muy usados en los SO, pero las llamadas al sistema que los implementan no se han estandarizado entre los diferentes SO.
El hardware para medir tiempo transcurrido e iniciar operaciones se denomina *temporizador de intervalos programables* y puede configurarse de modo que espere cierto tiempo y luego genere una interrupción. Algunos SO ofrecen una interfaz para

que los procesos de usuario utilicen temporizadores. Para ello el núcleo (o el driver del temporizador) mantiene una lista de interrupciones deseadas por sus propias rutinas y por solicitudes de usuario, ordenada según el tiempo de vencimiento. Cuando el temporizador interrumpe, el núcleo envía una señal al solicitante y vuelve a cargar el temporizador con el siguiente tiempo de vencimiento.
En la mayoría de las computadoras el reloj de hardware se construye con un contador de alta frecuencia. Este reloj no genera interrupciones pero permite medir con exactitud intervalos de tiempo.

12.3.4 E/S bloqueadora y no bloqueadora

Cuando una aplicación usa una llamada al sistema *bloqueadora*, la ejecución de la aplicación se suspende. La aplicación pasa de la cola de ejecución a una cola de espera, y luego de finalizada la llamada al sistema la aplicación se coloca otra vez en la cola de ejecución, donde recibe los valores devueltos por la llamada al sistema. A pesar de que las acciones de los dispositivos de E/S se realizan de forma asincrónica (tardan un tiempo impredecible) la mayor parte de los SO usan llamadas al sistema bloqueadoras, porque son mas fáciles de entender.
Una llamada al sistema no bloqueadora no detiene la ejecución de la aplicación por un largo tiempo, sino que regresa rápidamente con un valor de retorno. Una alternativa para una llamada al sistema no bloqueadora es una llamada al sistema asincrónica, que regresa de inmediato sin esperar a que termine la entrada y salida, y la aplicación sigue ejecutando su código. La diferencia que esta tiene con una llamada al sistema no bloqueadora es que la asincrónica avisa con una interrupción cuando termina, y la no bloqueadora no espera a que termine la E/S sino tiene un tiempo tope en el cual puede retornar todos, algunos o ninguno de los bytes solicitados.

12.4 Subsistemas de E/S del núcleo

Los núcleos ofrecen muchos servicios relacionados con la E/S.

12.4.1 Planificación de E/S

Se utiliza para mejorar el desempeño global del sistema y reducir el tiempo de espera promedio de E/S. Se utiliza una cola de solicitud para cada dispositivo, la cual es reacomodada por el planificador, para mejorar el tiempo de resupuesta.

12.4.2 Uso de buffers

Un buffer es un área de memoria en la que se almacenan datos mientras se transfieren entre dos dispositivos o entre un dispositivo y una aplicación. Encontramos tres usos fundamentales de los buffers:

- Diferencia de velocidades entre el productor y el consumidor de una corriente de datos (Modem – Disco)
- Se utiliza como un adaptador entre dispositivos tiene diferentes tamaños de transferencias de datos (Redes).
- Apoyar la semántica de copiado de E/S de aplicaciones. Esta garantiza que la versión de los datos que se escribió en el disco es la versión que existía en el momento en que la aplicación hizo la llamada al sistema, sin importar los cambios hechos después en el buffer. Para esto el SO copia los datos de la aplicación en un buffer, antes de devolver el control a la aplicación.

12.4.3 Uso de cachés

Es una región de memoria rápida que contiene copias de datos. La diferencia entre un buffer y un caché es que el buffer podría contener la única copia existente de un elemento de información mientras que un caché solo tiene una copia en almacenamiento más rápido de un elemento que ya existe en otro lado.

12.4.4 Uso de spool y reservación de dispositivos

Un spool es un buffer que contiene salidas para dispositivos, como una impresora, que no puede aceptar corriente de datos intercalados. El SO proporciona una interfaz de control que permite exhibir la cola, eliminar trabajos, suspender la impresión,

etc., además permiten coordinar salidas concurrentes, en dispositivos que no pueden multiplexar útilmente las operaciones de E/S de varias aplicaciones concurrentes.

12.4.5 Manejo de Errores

Los dispositivos y las transferencias de E/S pueden fallar de forma transitoria o permanente. Algunas veces los sistemas pueden compensar los fallos transitorios, pero pocas veces pueden con los permanentes.
Por lo general una llamada al sistema de E/S devuelve un bit de información acerca del estado de la llamada, para indicar si tuvo éxito o fracaso. En UNIX se emplea una variable entera llamada **errno** para devolver un código de error, indicando la naturaleza del fallo.

12.4.6 Estructuras de datos del núcleo

El núcleo necesita mantener información de estado acerca del uso de los componentes de E/S, esto se hace a través de varias estructuras de datos internas. La comunicación puede ser a través de estructuras de datos compartidas, o a través de transferencia de mensajes, que simplificando la estructura y aumentan la flexibilidad del sistema de E/S, a pesar que implican un gasto extra.

12.5 Transformación de solicitudes de E/S en operaciones de hardware

Para leer un archivo del disco, la aplicación hace referencia a los datos con un nombre de archivo, y el sistema de archivos, debe establecer la correspondencia entre ese nombre, y el espacio que se le asignó en los directorios.

Lo que sucede en los diferentes sistemas es:

- **MS-DOS:** El nombre se transforma en un número que indica la entrada de la tabla de acceso a archivos, y esa entrada indica los bloques de disco.
- **UNIX:** El nombre se transforma en el número de i-nodo, y el i-nodo correspondiente contiene la información de asignación de espacio.

12.6 Desempeño

La E/S es un factor importante para el desempeño del sistema, ya que se deben planificar procesos, ejecutar código de controladores de dispositivos, conmutaciones de contexto, etc.
Aunque se manejen interrupciones, cada interrupción implica un cambio de estado, etc. El tráfico de red también puede causar frecuentes cambios de contexto.

Solaris implementa un demonio llamado *telnet* que elimina las conmutaciones de contexto en la transferencia de caracteres en redes.
Otros sistemas utilizan *procesadores frontales*, para reducir la carga de interrupciones de la CPU principal.

Se pueden aplicar varios principios para mejorar la eficiencia de la E/S:

- Reducir el número de conmutaciones de contexto.
- Reducir el número de veces que hay que copiar datos en la memoria durante la transferencia entre el dispositivo y la aplicación.
- Reducir la frecuencia de interrupciones realizando transferencias grandes y utilizando controladores inteligentes y escrutinio.
- Aumentar la concurrencia empleando controladores DMA para encargarse del copiado sencillo de datos que de otro modo tendría que efectuar la CPU.

- Incorporar primitivas de procesamiento en Hardware para que puedan operar en controladores de dispositivos en forma simultanea con el funcionamiento de la CPU y los buses.
- Equilibrar el desempeño de la CPU, el subsistema de memoria, los buses, la entrada y salida (la sobrecarga de cualquiera de estas causara ociosidad en otras.

13 Estructura del almacenamiento secundario

13.1 Estructura de Discos

Los discos son los dispositivos de almacenamiento secundario más usados en los sistemas de computación modernos. Las unidades de disco modernas se direccionan como grandes arreglos unidimensionales de *bloques lógicos* (la unidad más pequeña de transferencia), que se hace corresponder secuencialmente con los sectores de disco (el sector 0 es el primero de la pista más exterior del cilindro). El tamaño usual del bloque es de 512 bytes.
De esta forma, es posible convertir un número de bloque lógico en una dirección de disco, aunque es una práctica difícil ya que:

- Pueden existir sectores defectuosos, pero la correspondencia oculta esto sustituyendolos por sectores de reserva de otro punto del disco.
- El número de sectores por pista no es constante, ya que cuanto más lejos está del centro, más larga es y puede contener más sectores. Por esta causa se organizaron en *cilindros* y el número de sectores por pista es constante dentro de ellos.

13.2 Planificación de discos

El sistema operativo debe usar eficientemente las unidades de disco, lo que implica tener un tiempo de acceso breve y un gran ancho de banda de disco. Definiciones:

- **Ancho de banda del disco:** Es el número de bytes transferidos dividido entre el tiempo total transcurrido entre la primera solicitud de servicio y la finalización de la última transferencia.
- **Tiempo de acceso:** Tiene dos componentes principales:
  - **Tiempo de Seek:** Tiempo que tarda el brazo del disco en mover las cabezas del cilindro que contienen el sector deseado.
  - **Latencia rotacional:** Tiempo adicional que el disco tarda en girar hasta que el sector deseado queda bajo la cabeza del disco.

Para mejorar el tiempo de acceso y el ancho de banda hay que planificar la atención de las solicitudes de E/S de disco en un orden apropiado.

13.2.1 Planificación FCFS (First come, first served)

Es la forma más sencilla de planificación. Es el algoritmo más justo pero no proporciona el servicio más rápido.

Ejemplo:

Una cola tiene solicitudes de E/S a bloques que están en los cilindros: 98 183 37 122 14 124 65 67
Si la cabeza del disco esta en el cilindro 53, primero se moverá del 53 al 98, luego al 187, 37… y así hasta al final, con un movimiento de cabeza de 640 cilindros. El problema con este algoritmo es que puede presentar una amplia oscilación por lo que lo hace bastante ineficiente.

13.2.2 Planificación SSTF (shortest seek time first)

Atiende todas las solicitudes cercanas a la posición actual de la cabeza antes de mover a la cabeza a la posición lejana para atender otras solicitudes.En el ejemplo la solicitud más cercana de la cabeza (53) es 65, luego 67, 37, 14, 98, 122, 124 y 183. Por lo que hay un movimiento total de cabeza de 236, mucho mejor que el FCFS. Pero esta planificación al igual que SJF puede causar inanición de algunas solicitudes, por lo que mejora FCFS pero no es óptima.

13.2.3 Planificación SCAN (algoritmo del elevador)

El brazo del disco parte de un extremo del disco y se mueve hacia el otro, atendiendo las solicitudes a medida que llega a cada cilindro, hasta llegar hasta el otro extremo del disco, donde se invierte la dirección de movimiento de la cabeza, continuando la atención.
En el ejemplo, suponemos que la dirección de movimiento de la cabeza se mueve de 53 a 0, por lo que primero se atiende a 37, luego 14, 65, 67, 98, 122, 124, 183.

13.2.4 Planificación C-SCAN (SCAN circular)

La planificación esta diseñada para darán tiempo de espera mas uniforme. Al igual que SCAN se mueve la cabeza de un extremo de disco a otro atendiendo las solicitudes en el camino, solo que cuando llega la otro extremo regresa de inmediato al principio del disco sin atender solicitudes. Esto se hace ya que luego que cambio de dirección hay pocas solicitudes inmediatamente después de la cabeza (debido a que hace poco que se atendieron esos cilindros) y la mayor densidad de solicitudes se encuentra en el otro extremo del disco.

13.2.5 Planificación LOOK

En la práctica, ni SCAN ni C-SCAN se implementan así, sino que el brazo solo llega hasta la última solicitud en cada dirección y luego cambia la dirección inmediatamente sin llegar al otro extremo del disco. En este caso la implementación es llamada LOOK Y C-LOOK respectivamente.

13.2.6 Selección de un algoritmo de selección de disco

La elección de un algoritmo depende en gran medida del número y los tipos de solicitudes, por lo que no hay un algoritmo más o menos efectivo. Por esta razón los algoritmos de planificación de disco se escriben como módulo independiente del SO, a fin de poder sustituirlo por un algoritmo diferente si es necesario. Por omisión usualmente se utilizan los algoritmos de SSTF y LOOK.
Todos los algoritmos descritos no consideran la latencia rotacional, que se puede tan grande como el tiempo de búsqueda promedio, ya que es difícil resolverla mediante planificación, porque los discos modernos no revelan la especificación física de los bloques lógicos.

13.3 Administración de Discos

13.3.1 Formateo de Discos

Antes de poder almacenar datos en el disco magnético es necesario dividirlo en sectores que el controlador de disco pueda leer y escribir, a este proceso se lo llama *formateo de bajo nivel* o *formateo físico*. El formateo llena el disco con una estructura de datos especial para cada sector, por lo general consiste en un cabecera, una área de datos y un apéndice. La cabecera y el apéndice tienen información que usa el controlador, así como un número de sector y un *código de corrección de errores* (ECC). Cada vez que se escribe un sector de datos durante la E/S normal el ECC se actualiza y se controla con un valor calculado a partir de todos los bytes del área de datos. El ECC contiene suficiente información como para detectar uno o dos bits erróneos. En muchos discos duros, cuando se les da formato de bajo nivel al disco, se puede especificar cuantos bytes de espacio de datos se debe dejar entre la cabecera y el apéndice de todos los sectores.
Formatear un disco con un tamaño de sector mayor implica que cada pista puede contener menos sectores, pero también hay que escribir menos cabeceras y apéndices en cada pista, lo que aumenta el espacio disponible para los datos de usuario.

Para poder utilizar el disco para almacenamiento de archivos primero hay que dividirlo en uno o más grupos de cilindros (particiones), tratándose a cada partición como si fuera un disco individual. El segundo paso es el *formateo lógico* (o creación de un sistema de archivos).

13.3.2 Sector de Arranque

Para que un computador empiece a funcionar necesita tener un programa inicial que ejecutar. El programa de *carga inicial de arranque* es sencillo y solamente asigna valores iniciales a todos los aspectos del sistema desde registro de CPU hasta los controladores de dispositivos y pone en marcha el SO. El programa encuentra el núcleo en el disco, lo carga en memoria y salta a una dirección inicial para comenzar la ejecución del SO.
Casi todas las computadoras, el programa de carga inicial de arranque en la memoria ROM, cuyo propósito es traer a memoria y ejecutar un programa de carga inicial completo del disco (disco de booteo).
El programa de carga inicial completo, se almacena en una partición llamada *bloque de arranque*, en una posición fija del disco.

13.3.3 Bloques defectuosos

Los discos son propensos a fallos, dependiendo del disco y el controlador empleados, los *bloques defectuosos* se manejan de diferentes maneras.
En los discos sencillos (disco IDE) estos bloques se manejan manualmente. Son detectados y aislados al formatear o al hacer un scandisk, escribiendo un valor especial en la FAT, indicando que no pueden ser usados. Los datos que residían en los bloques defectuosos por lo general se pierden.
En los discos más avanzados (SCSI) tienen procedimiento de recuperación de bloques defectuosos más inteligentes. La lista se inicia durante el formateo de bajo nivel en la fabrica y se actualiza durante toda la vida del disco. El formateo de bajo nivel aparta sectores de reserva que el SO no ve, y se le puede pedir el controlador que sustituya cada sector defectuoso lógicamente por uno de los sectores de reserva.
Los discos se formatean guardando sectores de respaldo en cada cilindro, de manera que si debo reemplazar un sector, su reemplazo esté lo más cerca posible (dentro del mismo cilindro).

13.4 Administración del espacio de Intercambio

La memoria virtual utiliza un espacio de disco como una extensión de la memoria principal y puesto que el disco es mucho mas lento que la memoria, debo administrar ese espacio de intercambio a fin de lograr el mejor desempeño posible.

13.4.1 Uso del espacio de Intercambio

La cantidad de espacio de intercambio de un sistema puede variar dependiendo de la memoria física, la cantidad de memoria virtual que esta manejando y la forma en que la forma virtual se usa.

13.4.2 Ubicación del espacio de intercambio

Hay dos lugares en los que puede residir un espacio de intercambio: en el sistema de archivos normal o en una partición de disco aparte.
El primer sistema es fácil de implantar pero ineficiente debido que hay navegar la estructura de directorio consumiendo un tiempo excesivo.
En el segundo sistema se usa un administrador de espacio de intercambio exclusivo (sin el sistema de archivos) para asignar y liberar los bloques, el cual emplea algoritmos que han sido optimizados en el aspecto de rapidez. La desventaja es que crea una cantidad fija de espacio de intercambio durante la partición del disco, y para añadir mas espacio hay que volver a particionar el disco.
En algunos sistemas operativos permiten utilizar ambos métodos. Ej: Solaris 2.

13.5 Confiabilidad de los Discos

Se han propuestos varias mejoras en las técnicas de uso de discos. Estos métodos implican el uso de múltiples discos que trabajan en forma cooperativa. Cada bloque de datos se divide en sub bloques y se almacena un sub bloque en cada disco. Por lo que el tiempo para transferir un bloque a la memoria se reduce ya que los discos transfieren sus sub bloques en paralelo. Esta organización se llama *conjunto redundante de discos independientes* (RAID), y mejora la confiabilidad del sistema de almacenamiento porque se almacenan datos redundantes. Tipos de RAID:

- **Espejos:** Mantiene una copia duplicada de cada disco. Es costosa porque se necesita el doble de discos para almacenar la misma cantidad de datos.
- **Bloques de paridad intercalados:** No es tan redundante. Una fracción de disco se usa para contener bloques de paridad. Si ocurre un error, los bits perdidos se pueden recalcular. En este caso, se puede bajar el desempeño, ya que cada escritura implica recalcular y escribir el subbloque de paridad correspondiente. Los sub bloques de paridad deben estar distribuidos en los discos (no en uno solo), a fin de repartir la carga de trabajo.

13.6 Implementación de almacenamiento estable

Se debe contar con un mecanismo que garantice que un fallo durante una actualización de datos, no dejará todas las copias en estado dañado, y que al recuperarse del fallo, queden todas en estado consistente.
La escritura de disco puede tener tres estados:

- **Finalización con éxito:** Los datos se escribieron correctamente.
- **Fallo parcial:** Ocurrió un fallo a la mitad de la transferencia. Solo algunos sectores están actualizados, y el que se estaba escribiendo cuando ocurrió el fallo, contiene valores erróneos.
- **Fallo total:** Ocurrió antes de iniciarse la escritura a disco, de manera que los valores anteriores permanecen intactos.

Para garantizar una correcta escritura, debemos mantener dos bloques físicos por cada bloque lógico. Escribiendo primero en un bloque físico, luego en el segundo, y cuando los dos se escribieron correctamente, se da por terminada la operación.
Durante la recuperación se examinan los dos bloques, y hay error si no coinciden. En ese caso sustituimos el contenido del primer bloque, por el del segundo, a fin de dejar los datos como estaban antes de comenzar la escritura.

14 Estructura de almacenamiento terciario

14.1 Dispositivos de almacenamiento terciario

El bajo costo es la característica distintiva del almacenamiento terciario, por esto se componen de *medios removibles* (CD-ROM, discos flexibles, etc).

14.1.1 Discos removibles

Los *diskettes* son un ejemplo de discos magnéticos removibles. Contienen un disco delgado y flexible recubierto con material magnético encerrado en un estuche plástico protector. Sin embargo otros discos magnéticos removibles pueden contener más de un Gigabyte, siendo casi tan rápidos como los discos duros pero su superficie tiene más riesgo a dañarse por rayones.
También están los discos *magneto-ópticos* que graban los datos en un plato rígido recubierto con material magnético, encerrado en una gruesa capa de plástico y vidrio, lo que los hace más resistentes.
Los discos *ópticos*, no necesitan magnetismo, sino que emplean materiales que se pueden alterar con luz láser para que tengan puntos oscuros o brillantes. En cada punto se almacena un bit. Los discos pueden ser:

- **Discos de cambio de fase:** Están recubiertos con un material que puede solidificarse en un estado cristalino o amorfo, ambos estados reflejan la luz láser con diferente intensidad.
- **Discos de colorante polímero:** Están recubiertos de un plástico que contiene un colorante que absorbe la luz láser. El

láser puede calentar un punto de modo de que se forme una protuberancia, y también puede calentar una protuberancia de modo de que vuelva a su forma normal.

Los discos magneto-óptico son los mas usados a pesar de su alto costo y bajo desempeño.
Los discos nombrados anteriormente son discos de leer-escribir, pero también se encuentran los discos WORM, que son aquellos en los que se puede escribir una vez y leer muchas veces, estos discos se consideran duraderos y confiables. Luego están los discos solo de lectura como CD-ROM y DVD, que vienen de fábrica con los datos pregrabados, estos también son duraderos.

14.1.2 Cintas

Son el medio mas económico para fines que no requieren acceso aleatorio rápido, comúnmente se utilizan para guardar copias de respaldo de datos de disco, y también para guardar enormes volúmenes de datos.

14.2 Tareas de sistema operativo

El sistema operativo tiene como tareas importantes administrar los dispositivos físicos y presentar una abstracción de máquina virtual a las aplicaciones.

14.2.1 Interfaz con las aplicaciones

La mayor parte de los SO manejan los discos removibles casi igual que los fijos. Cuando un disco nuevo se inserta en la unidad (se “monta”) es preciso formatearlo y luego generar un sistema de archivos vacío en él (al igual que en un disco duro).
Las cintas se manejan de manera diferente, ya que cuando una aplicación la utiliza, la unidad de cinta queda reservada para el uso exclusivo de esa aplicación, hasta que la aplicación termina o cierra el dispositivo de cinta. Esto es razonable ya que intercalar accesos aleatorios a cintas desde una aplicación causaría hiperpaginación.
Dentro de las operaciones básicas de cintas no se encuentra buscar, sino que esta ubicar. Donde ubicar coloca la cinta en un bloque lógico especifico en vez de una pista completa. Si la cinta no esta totalmente llena no es posible ubicarse en el espacio vacío que esta mas allá de área escrita, ya que todas las unidades de cinta tienen bloques de tamaño variable, y el tamaño de cada bloque es determinado sobre la marcha, o sea cuando se escribe ese bloque. Además si se encuentra un área defectuosa durante la escritura, se pasa por alto y el bloque se escribe otra vez, las posiciones y números de bloque lógicos todavía no se han determinado. También tiene operaciones para moverse en los bloques, por ejemplo dos bloques atrás, etc.
La unidad de cinta implementa la anexión colocando una marca de fin de cinta (EOT) después de escribir bloques, luego para anexar nuevos datos se sobrescribe esta marca, y se coloca una nueva al final de los bloque recién escritos.

14.2.2 Nombres de archivos

Los SO actuales generalmente dejan sin resolver el problema del espacio de nombre en los medios removibles, y confían en que las aplicaciones y los usuarios averiguarán por su cuenta como acceder a los datos e interpretarlos. A pesar de esto existen algunos medios removibles que están bien estandarizados y todos los computadores los usan por igual.

14.2.3 Gestión de almacenamiento jerárquico

Se utilizan medios de almacenamiento terciario para respaldar la memoria primaria y el almacenamiento secundario. Generalmente estos medios son más económicos, pero también más lentos.

14.3 Cuestiones de desempeño

14.3.1 Rapidez

Hay dos aspectos importantes: el ancho de banda y la latencia.

- **Ancho de banda sostenido:** Tasa de datos promedio durante una transferencia grande (numero de bytes/tiempo de transferencia).
- **Ancho de banda efectivo:** Promedio durante todo el tiempo de E/S, incluido el tiempo para buscar o ubicarse.

14.3.2 Confiabilidad

Una unidad de disco duro será mas confiable que una unidad de disco removible o cinta, y un cartucho óptico probablemente será mas confiable que un disco magnético o una cinta magnética. Sin embargo en un disco duro un aterrizaje de cabeza generalmente destruye los datos, mientras que los fallos en una unidad de cinta o de disco óptico muchas veces no dañan el cartucho de datos.

14.3.3 Costo

El almacenamiento terciario representa un ahorro en el costo, solo cuando el número de cartuchos es considerablemente mayor que el número de unidades.

19 Protección

Los diversos procesos de un sistema operativo se deben proteger de las actividades de los demás procesos. Existen mecanismos que garantizan que solo los procesos autorizados por el SO, puedan operar con los distintos recursos.
La *protección* se refiere a un mecanismo para controlar el acceso de procesos o usuarios al sistema, controles sobre ellos, etc.
La *seguridad* es una medida de confianza que tenemos en que se preservará la integridad del sistema.

19.1 Objetivos de la protección

Originalmente se concibió para que los usuarios compartieran recursos sin peligro en sistemas multiprogramados. Actualmente evolucionaron para aumentar la confiabilidad de sistemas que usan recursos compartidos.

Razones para dar protección:

- Evitar que usuarios malintencionados violen restricciones de acceso.
- Asegurar que cada componente de un programa use los recursos respetando las políticas de uso de ellos.

La protección puede detectar errores en las interfaces entre subsistemas, evitando la contaminación de un subsistema sano, con otro que no funciona correctamente. Existen mecanismos para distinguir entre accesos autorizados y no autorizados.

Tipos de políticas de uso de recursos:

- Determinadas por el diseño del sistema
- Formuladas por los administradores del sistema
- Definidas por usuarios para proteger sus propios archivos o programas

Las políticas pueden cambiar en el tiempo o dependiendo de la aplicación, por lo tanto el sistema debe proveer de herramientas al programador para que pueda modificarlas o utilizarlas.

19.2 Dominios de protección

Un sistema consta de procesos y objetos, éstos últimos tanto de hardware (memoria, periféricos, etc.), como de software (archivos, semáforos, etc.). Cada objeto tiene un identificador, y se accede a él con operaciones bien definidas que dependen del objeto (CPU: Ejecución, Memoria: lectura, escritura, etc.).

Un proceso debe acceder a los recursos para los cuales está autorizado, y en un momento dado solo puede acceder a aquellos que necesite para llevar a cabo su tarea. Esto es conocido como el principio de necesidad de conocer.

19.2.1 Estructura de dominios

Un proceso opera dentro de un *dominio de protección*, que especifica los recursos a los que puede acceder. El dominio define el conjunto de objetos, y el tipo de operaciones que se pueden invocar sobre ellos. La capacidad para ejecutar una operación con un objeto, se llama *derecho de acceso* y un dominio, es una colección de derechos de acceso de la forma:

<*nombre_de_objeto, conjunto_de_derechos*>

Tipos de asociación entre proceso y dominio:

- **Estática:** El conjunto de recursos de un proceso no cambia durante la ejecución del mismo, y queremos respetar el principio de necesidad de conocer, por lo tanto necesitamos un mecanismo para cambiar el contexto de un dominio, para reflejar los derechos de acceso mínimos necesarios. Por ejemplo: Un proceso tiene dos fases. En una requiere escritura y en otra lectura. Si proporcionamos al proceso un dominio que incluye los dos accesos, proporcionamos más derechos de los necesarios a cada una de las fases.
- **Dinámica:** Se cuenta con un mecanismo que cambia de un dominio a otro. También se podría modificar el contenido del dominio, o en su defecto, crear un nuevo dominio con el contenido deseado, y cambiando a él.

Formas de establecer dominios:

- **Usuario:** El conjunto de objetos a los que se puede acceder depende de la identidad del usuario. Hay conmutación de dominio cuando se cambia de usuario (cierre y apertura de sesión).
- **Proceso:** El conjunto de objetos a los que se puede acceder depende de la identidad del proceso. La conmutación de dominio corresponde a que un proceso envía un mensaje a otro proceso y espera su respuesta.
- **Procedimiento:** El conjunto de objetos a los que se puede acceder corresponde a variables locales definidas dentro del procedimiento. La conmutación de dominio ocurre cuando se invoca a otro procedimiento.

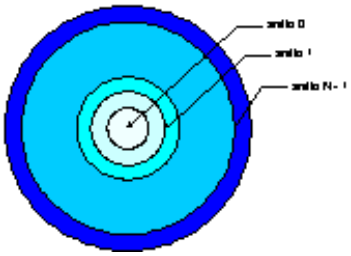
19.2.2 Ejemplos

El modo dual (usuario – monitor) es una forma de proteger el SO. Cada uno tiene su propio dominio. En un sistema multiusuario no alcanza con esta protección puesto que se debe proteger un usuario de otro.

19.2.2.1 UNIX

Asocia un dominio a cada usuario, y cambiar de dominio significa cambiar temporalmente la identificación de usuario.
Cada archivo tiene asociados una identificación de propietario y un bit de dominio. Cuando un usuario con user-id = A, comienza a ejecutar un archivo de propiedad de B, con el bit de dominio apagado, el user-id del proceso se hace igual a A. Si el bit de dominio está prendido, el user-id del usuario se hace igual al del propietario del archivo, hasta que el proceso termina.
Esto se usa para poner a disposición e usuarios, los recursos privilegiados. Por ejemplo, el usuario puede tomar momentáneamente el user-id “root” para usar un recurso, pero hay que controlar que no pueda crear un archivo con user-id “root” y el bit encendido, porque podría convertirse en “root”.
Hay sistemas más restrictivos que no permiten cambiar los identificadores de usuario. Se deben usar mecanismos especiales para acceder a recursos especiales como “demonios”.

19.2.2.2 Multics



Los dominios de protección se organizan jerárquicamente en una estructura de anillos. Cada anillo corresponde a un dominio. El proceso que se ejecuta en el dominio D_0 es el que más privilegios tiene.
Tiene un espacio de direcciones segmentado donde cada segmento es un archivo y está asociado a un anillo. A cada proceso se le asocia un número de anillo, y puede acceder a segmentos asociados a procesos con menos privilegios (mayor número de anillo), pero no a los segmentos más privilegiados.
Se produce una conmutación de contexto cuando un proceso invoca un procedimiento de otro anillo, y esa conmutación es controlada por el SO. Para este control, el anillo cuenta con los campos:

- **Intervalo de acceso:** (b_1, b_2) y $b_1 \leq b_2$.
- **Limite:** b_3 tal que $b_3 > b_2$
- **Lista de puertas:** Identifica puertas en las que se pueden invocar segmentos.

Un proceso en el anillo i, puede ejecutar segmentos de un intervalo de acceso (b_1, b_2), solo si $b_1 \leq i \leq b_2$. Si no se cumple se efectúa una trampa y se hacen nuevos controles:

- $i < b_1$: Se permite la llamada porque el anillo que busco es menos privilegiado.
- $i > b_2$: Solo se permite la llamada si $b_3 \leq i$. La llamada se destina a una “puerta”, permitiendo que los procesos con derechos de acceso limitado, puedan invocar procedimientos más privilegiados, pero de manera controlada.

Desventaja: No permite el cumplimiento de la necesidad de conocer.

Los mecanismos de protección también dependen de la necesidad de seguridad o performance. En una computadora en la que se procesan calificaciones de estudiantes, y a la que acceden estudiantes, se justifica un complejo sistema de seguridad, pero en una que solo es necesaria para triturar números, tiene más peso la performance.

19.3 Matriz de acceso

Las decisiones de política relativas a la protección se pueden implementar con una matriz de acceso. Las filas representan dominios, y las columnas objetos. Cada entrada es un conjunto de derechos de acceso.

Dominio \ Objeto	F	F	F	Impresora
D_1	leer		escribir	
D_2		leer		imprimir

Todos los procesos que se ejecutan en el dominio D_1, pueden leer el archivo F_1, y escribir el F_3. Los que se ejecutan en D_2, pueden leer el archivo F_2, e imprimir.

Conmutación

Si incluimos también a la matriz como un objeto, podemos darle privilegios solo a algunos dominios para modificarla. De la misma forma, podemos controlar la conmutación de dominios.

Dominio \ Objeto	F	F	F	Impresora	D_1	D_2
D_1	leer		escribir			conmutar
D_2		leer		imprimir		

Un proceso en el dominio D_1, puede conmutar al dominio D_2.

Copia

Se puede dar la capacidad de copiar derechos, añadiendo un (*) al derecho de acceso. Se puede copiar el derecho de acceso solo dentro de la misma columna (en el mismo objeto).
Variantes:

- **Transferencia:** Se copia el acceso (j, k) al acceso (i, k) y se borra el acceso (j, k).
- **Limitación:** El derecho (R*) se copia de (j, k) a (i, k) creando el derecho R, y no R*, por lo tanto un proceso en el dominio l, no puede copiarlo.

También el sistema podría elegir que derecho darle a la copia, si el de “copia”, “trasferencia” o “copia limitada” colocando diferentes valores en la matriz.

Se necesita un mecanismo que permita añadir o quitar derechos. Para eso se le asigna el derecho de dueño a uno de los dominios que utiliza el objeto y solo él pede asignar o quitar derechos.
Se requiere un mecanismo que cambie derechos en las filas, y es el de “control”.

19.4 Implementación de la matriz de acceso

La mayor parte de las entradas de la matriz están vacías.

19.4.1 Tabla global

Consiste en un conjunto de tripletas ordenadas:

<*dominio, objeto, conjunto_de_derechos*>

Si ejecuto una operación M, con un objeto O_j, en el dominio D_i, busco la entrada $<D_i, O_j, R_k>$ $M \in R_k$. Si no existe, se genera error.

Desventajas: La tabla es grande, por lo tanto no puede estar en memoria principal. Además hay información repetida, ya que si todos los dominios usan de la misma forma un mismo objeto, debo tener una entrada de la tabla para cada dominio.

19.4.2 Lista de acceso para objetos

Cada columna (objeto) es una lista de acceso, que contiene los pares <*dominio, conjunto_de_derechos*> (conjunto_de_derechos <> ∅).
En este caso se puede implementar un conjunto de derechos por omisión, que contiene todo los objetos a los que puedo acceder desde cualquier dominio.

19.4.3 Lista de capacidades para dominios

Es igual que en el caso anterior, pero para los dominios. Cada dominio tiene una lista de capacidades (objetos y operaciones). El usuario puede acceder a estas listas indirectamente, ya que están protegidas por el sistema.

Protección de capacidades:

- Cada objeto tiene una etiqueta para denotar su tipo (puntero, booleano, valor no inicializado). No son directamente accesibles al usuario.
- El espacio de direcciones de un programa se puede dividir:
  - Una parte accesible con datos e instrucciones normales.
  - Otra parte con la lista de capacidades accesibles

19.4.4 Un mecanismo de cerradura y llave

Es un término medio entre las dos listas (capacidades y acceso). Cada objeto tiene una lista de patrones de bit llamada *cerradura*, y cada dominio tiene otra llamada *llave*.
Para que un proceso que se ejecuta en un dominio pueda acceder a un objeto, el dominio debe tener la llave adecuada.

19.4.5 Comparación

Cuando un usuario crea un objeto, establece sus derechos, pero es difícil determinar el conjunto de derechos para cada dominio. Las listas de capacidades son útiles para localizar información de un proceso en particular pero no se corresponden con las necesidades del usuario.
El mecanismo de cerradura y llave, es un término medio. Puede ser eficaz y flexible, dependiendo de la longitud de las llaves, y los privilegios de acceso se pueden revocar para un dominio, simplemente cambiando su llave.
La mayor parte de los sistemas usa una combinación entre listas de acceso y capacidades.

19.5 Revocación de derechos de acceso

Tipos:

- **Inmediata o diferida**
- **Selectiva o general:**
  - *General:* Al revocar un derecho de acceso afecto a todos los usuarios con ese derecho.
  - *Selectiva:* Puedo seleccionar usuarios para revocarlo.
- **Parcial o total:**
  - *Total:* Revoco todos los derechos de acceso a un objeto.
  - *Parcial:* Revoco un subconjunto.
- **Temporal o permanente:**
  - *Temporal:* Se puede revocar el derecho y luego obtenerlo nuevamente.
  - *Permanente:* Si lo revoco, no puedo volverlo a obtener.

Con un sistema de lista de acceso la revocación es fácil. Es inmediata y puede ser general, selectiva, etc. Con listas de capacidades debo hallarlas en todo el sistema antes de revocarlas.

Esquemas de revocación:

- **Readquisición:** Periódicamente se eliminan capacidades de cada dominio. El proceso tiene que tratar de readquirirla para usarla, y si fue revocada, no la puede adquirir.
- **Retropunteros:** Cada objeto tiene punteros asociados a sus capacidades. Se puede seguir esos punteros para revocar o modificar capacidades. Su implementación es costosa.
- **Indirección:** Las capacidades no apuntan directamente al objeto sino a una entrada única de una tabla global. Para revocar, busco la entrada en la tabla global y la elimino. No permite revocación selectiva.
- **Claves:** La clave se define al crear la capacidad y el proceso que posee dicha capacidad no puede modificarla. Debo cambiar la clave maestra de cada objeto con un “set_key” para revocar los derechos. No permite revocación selectiva.

19.6 Sistemas basados en capacidades

19.6.1 Hydra

Es flexible. Proporciona un conjunto fijo de posibles derechos de acceso que el sistema conoce, y proporciona a un usuario del sistema de protección, un mecanismo para declarar nuevos derechos. Las operaciones sobre objetos, son tratadas también como objetos, y se accede a ellas a través de capacidades.
Otro concepto interesante es la *amplificación de derechos* que certifica un procedimiento como *confiable* y le permite acceder a objetos inaccesibles, a nombre de otro procedimiento que si puede accederlos. Estos derechos no son universales, y están restringidos.
Ej: Un procedimiento invoca una operación P, la cual es amplificada. La operación utiliza sus derechos nuevos, y cuando termina, el procedimiento vuelve a tener los derechos normales.

Hydra diseñó protección directa contra el problema de *subsistemas que sospechan uno del otro*. Cuando varios usuarios invocan un programa de servicios, asumen el riesgo de que falle, perdiendo datos, o que retenga los datos y puedan ser usados posteriormente sin autorización.

19.6.2 Sistema Cambridge CAP

Es más sencillo que Hydra. Proporciona también protección segura a datos de usuario.
Capacidades:

- **Capacidad de datos:** Sirven para obtener acceso a objetos y tienen los derechos estándar (leer, escribir, etc.). Está en el micro código de CAP.
- **Capacidad de software:** Corre por cuenta de un procedimiento privilegiado. Cuando es invocado, él mismo obtiene los derechos de leer y escribir.

19.7 Protección basada en el lenguaje

La protección se logra con la ayuda del núcleo del SO que valida los intentos de acceso a recursos. El gasto de inspeccionar y validar todos los intentos de acceso a todos los recursos es muy grande, por lo tanto debe ser apoyada por hardware.
Al aumentar la complejidad del SO, se deben refinar los mecanismos de protección. Los sistemas de protección, no solo se preocupan de si puedo acceder a un recurso, sino también de cómo lo accedo, por lo tanto los diseñadores de aplicaciones deben protegerlos, y no solo el SO.
Los diseñadores de aplicaciones mediante herramientas de los lenguajes de programación pueden declarar la protección junto con la tipificación de los datos.

Ventajas:

- Las necesidades de protección se declaran sencillamente y no llamando procedimientos del SO.
- Las necesidades de protección pueden expresarse independientemente de los recursos que ofrece el SO.
- El diseñador no debe proporcionar mecanismos para hacer cumplir la protección.
- Los privilegios de acceso están íntimamente relacionados con el tipo de datos que se declara.

Diferencias entre las distintas formas de protección:

- **Seguridad:** La obligación de cumplimiento por núcleo ofrece un grado de seguridad que el código de seguridad ofrecido por el compilador.
- **Flexibilidad:** La flexibilidad de la implementación por núcleo es limitada. Si un lenguaje no ofrece suficiente flexibilidad, se puede extender o sustituir, perturbando menos cambios en el sistema que si tuviera que modificarse el núcleo.
- **Eficiencia:** Se logra mayor eficiencia cuando el hardware apoya la protección.

La especificación de protección en un lenguaje de programación permite describir en alto nivel las políticas de asignación y uso de recursos.
El programador de aplicaciones necesita un mecanismo de control de acceso seguro y dinámico para distribuir capacidades a los recursos del sistema entre los procesos de usuario.
Las construcciones que permiten al programador declarar las restricciones tienen tres operaciones básicas:

- Distribuir capacidades de manera segura y eficiente entre procesos clientes.
- Especificar el tipo de operaciones que un proceso podría invocar en un recurso asignado.
- Especificar el orden en que un proceso dado puede invocar las operaciones de un recurso.

20 Seguridad

La protección es estrictamente un problema interno, la seguridad en cambio no solo requiere un sistema de protección adecuado, sino también considerar el entorno externo donde el sistema opera.
La protección no es útil si la consola del operador está al alcance de personal no autorizado. La información debe protegerse de accesos no autorizados, destrucción, alteración mal intencionada o inconsistencias.

20.1 El problema de la seguridad

Un sistema es seguro si los recursos se usan y acceden como es debido en todas las circunstancias. Esto no siempre es posible pero se debe contar con mecanismos que garanticen que las violaciones de seguridad sean un suceso poco común.
Formas de acceso mal intencionado:

- Lectura no autorizada de datos (robo de información).
- Modificación no autorizada de datos.
- Destrucción no autorizada de datos.

El hardware del sistema debe proporcionar protección que permita implementar funciones de seguridad. MS-DOS casi no ofrece seguridad, y sería relativamente complejo agregar nuevas funciones.

20.2 Validación

Se deben identificar los usuarios del sistema.

20.2.1 Contraseñas

Cuando el usuario se identifica con un identificador, se le pide una contraseña, y si ésta coincide con la almacenada en el sistema, se supone que el usuario está autorizado. Se usan cuando no se cuenta con sistemas de protección más completos.

20.2.2 Vulnerabilidad de las contraseñas

El problema es la dificultad de mantener secreta una contraseña. Para averiguarla se puede usar fuerza bruta, usando todas las combinaciones posibles de letras, números y signos de puntuación, o probando con todas las palabras del diccionario.
El intruso podría estar observando al usuario cuando la digita o “husmeando” en la red, o sea, viendo los datos que el usuario transfiere a la red, incluidos identificadores y contraseñas.
Si el sistema elige la contraseña, puede ser difícil de recordar, y el usuario la anota, si la elige el usuario, puede ser fácil de adivinar.
Algunos sistemas envejecen las contraseñas, obligando al usuario a cambiarlas cada determinado tiempo. Se controla que el usuario no vuelva a usar una contraseña que ya usó antes, guardando un historial de contraseñas.

20.2.3 Contraseñas cifradas

El sistema UNIX usa cifrado para no tener que mantener en secreto su lista de contraseñas. Cada usuario tiene una contraseña y el sistema contiene una función, extremadamente difícil de invertir, pero fácil de calcular, que codifica las contraseñas y las almacena codificadas. Cuando el usuario presenta la contraseña, se codifica y compara con las ya codificadas. De esta forma no es necesario mantener en secreto el archivo con contraseñas.

Problemas:

- Se puede obtener una copia del archivo de contraseñas y comenzar a cifrar palabras hasta descubrir una correspondencia (ya que el algoritmo de cifrado de UNIX es conocido). Las nuevas versiones de UNIX ocultan el archivo.

- Muchos sistemas UNIX solo tratan los primeros 8 caracteres como significativos, quedando pocas opciones de contraseña.
- Algunos sistemas prohíben el uso de palabras de diccionario como contraseña, para que sea más difícil de adivinar.

20.3 Contraseñas de un solo uso

Se pueden usar contraseñas algorítmicas, donde el sistema y el usuario comparten un secreto. El sistema proporciona una semilla, el usuario con el algoritmo y la semilla introduce la contraseña, y el sistema la comprueba (ya que conoce el algoritmo del usuario). La contraseña es de un solo uso, por lo tanto, cualquiera que la capte, no podría utilizarla luego.

20.4 Amenazas por programas

20.4.1 Caballo de Troya

Muchos sistemas permiten que un usuario pueda ejecutar programas de otro usuario. Estos programas podrían aprovechar los derechos que le proporciona el dominio del usuario ejecutante y abusar de ellos. Un segmento de código que abusa de su entorno es llamado *caballo de Troya*.

20.4.2 Puerta secreta (Trap door)

El diseñador del programa podría dejar una puerta, por la cual entra salteándose todos los protocolos de seguridad. Se podría generar una puerta ingeniosa en un compilador, que coloque puertas en todos los programas que compile.
Una puerta en un sistema es difícil de detectar por la cantidad de código que el sistema posee.

20.5 Amenazas al sistema

20.5.1 Gusanos

El gusano engendra copias de si mismo ocupando los recursos del sistema, impidiendo que sean usados por otros procesos. En las redes son muy potentes, ya que pueden reproducirse entre los sistemas, y paralizar toda la red.

Gusano de Morris:

Causó en 1988 pérdidas millonarias a través de Internet. Era auto-replicante, para distribuirse rápidamente, además características de UNIX, le permitieron propagarse por todo el sistema. El gusano aprovechó defectos en rutinas de seguridad de UNIX, para obtener accesos no autorizados a miles de sitios interconectados. Ver ejemplo de página 631. Interesante pero inútil.

20.5.2 Virus

También están diseñados para extenderse hacia otros programas, y pueden producir daños graves en un sistema, como modificar o destruir archivos y causar caídas en programas o sistemas. Un gusano es un programa completo y autónomo; un virus es un fragmento de código incrustado en un programa legítimo.
Los computadores multiusuario no son tan vulnerables, porque protegen los programas ejecutables contra escritura, aún así es fácil su infección.

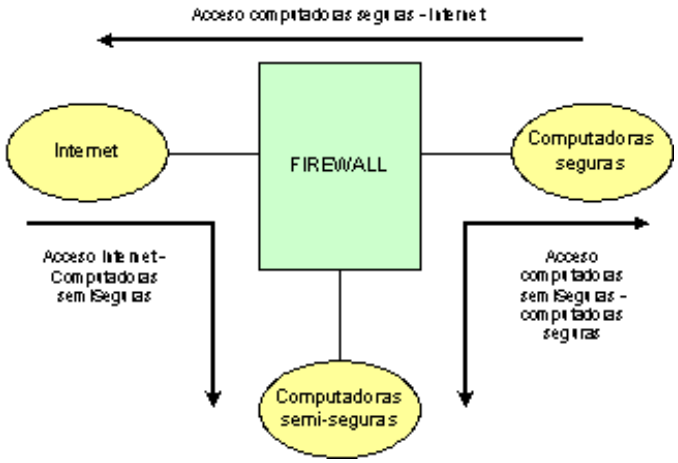
Antivirus: La mayor parte solo actúan sobre virus conocidos, examinando los programas del sistema en busca de instrucciones específicas que sabe que contiene el virus.

20.6 Vigilancia de amenazas

Se buscan patrones de actividad sospechosos. Las técnicas son:

- Cuenta de las veces que se proporcionan contraseñas incorrectas en el login
- Llevar una *bitácora de auditoría* que registra la hora, el usuario y los accesos a objetos, para determinar los problemas, pero estas bitácoras pueden crecer mucho, ocupando demasiados recursos en el sistema.
- Explorar el sistema periódicamente en busca de “agujeros” de seguridad, que se pueden realizar en el computador cuando hay poco tráfico, no como en el caso de las bitácoras. Puede examinar:
  - Contraseñas fáciles de adivinar.
  - Programas setuid no autorizados (si el sistema maneja el mecanismo).
  - Programas no autorizados en directorios del sistema.
  - Procesos cuya ejecución tiene una duración inusual.
  - Protecciones inapropiadas de directorios de usuario y sistema.
  - Protecciones inapropiadas de archivos de datos, contraseñas, drivers, o el núcleo del sistema.
  - Caballos de troya.

Los computadores en red son susceptibles a ataques por la gran cantidad de ataques remotos posibles.
Una ayuda de protección en red son los firewalls. Un firewall es un computador que se coloca entre lo confiable y lo no confiable, limita el acceso por red, y lleva una bitácora.



20.7 Cifrado

Por la red se transmite gran cantidad de información que no se desea que sea descubierta. Para lograr esto, se sigue el siguiente mecanismo:

- La información se cifra (codifica) de su forma comprensible original (texto limpio) a una forma interna (texto cifrado). La forma interna se puede leer pero no es comprensible.
- El texto cifrado se puede almacenar en archivos legibles y enviarse por canales desprotegidos.
- Para entender el texto, el receptor debe descifrarlo (decodificarlo) para obtener el texto limpio.

El reto es crear esquemas de cifrado imposibles de romper.
Hay varios métodos, pero uno de ellos consiste en tener un algoritmo de cifrado general E, uno de descifrado general D, y una o más claves secretas.
E_k y D_k son los algoritmos para la clave k.

Para el mensaje m se debe cumplir:

1. $D_k(E_k(m)) = m$
2. D_k, E_k se calculan de forma eficiente.
3. Se debe mantener en secreto la clave y no los algoritmos.

Se creó el DES, estándar de cifrado de datos, que contiene el problema de distribución de claves, ya que hay que enviar secretamente las claves.
Una solución a esto es usando clave pública. Cada usuario tiene dos claves, una pública y una privada, y para comunicarse solo basta con saber la pública del otro.

Algoritmo:

Clave pública = (e, n)
Clave privada = (d, n)

El mensaje se representa con un entero y se divide en submensajes de largo 0.. (n – 1)

$E(m) = m^e \bmod n = C$
$D(C) = C^d \bmod n$

n se calcula como el producto de dos primos p y q al azar (n = p x q); d es al azar, y satisface mcd [d, (p-1) x (q-1)] = 1; e se calcula tal que e x d mod [(p-1) x (q-1)] = 1.
Si es necesario, ver ejemplo en página 638.

20.8 Clasificación de seguridad de los computadores

El total de sistemas de computación que obligan al cumplimiento correcto de políticas de seguridad son llamados *Base de computador confiable (TCB)*.
Se clasifican en A, B, C y D:

- **D – Protección mínima:** Son sistemas evaluados que no satisfacen los requisitos de seguridad de las otras clases. Ej: MS-DOS y Windows 3.1.
- **C – Protección discrecional:** Se contabilizan los usuarios y sus acciones mediante auditorías. Tiene dos niveles:
  - **C1:** Permite que los usuarios puedan proteger información privada y evitar que otros la lean o destruyan. Abarca la mayor parte de las versiones de UNIX. La TCB de C1 controla el acceso entre usuarios y archivos permitiendo compartir objetos entre individuos nombrados, o grupos definidos. El usuario debe identificarse antes de iniciar cualquier actividad que la TCB deba mediar.
  - **C2:** Une a los requisitos de C1, un control de acceso a nivel individual, y el administrador del sistema puede auditar de forma selecta las acciones de cualquier usuario conociendo su identidad. Hay algunas versiones de UNIX que certifican el nivel C2.
- **B – Protección obligatoria:** Tienen todas las propiedades de C2, y además “pegan” etiquetas de confidencialidad a cada objeto.
  - **B1:** Mantiene la etiqueta de seguridad en cada objeto del sistema la cual es usada para tomar decisiones relacionadas con el control de acceso obligatorio. Maneja niveles de seguridad jerárquicos, de manera que usuarios pueden acceder a cualquier objeto con etiqueta de confidencialidad igual o menor.
  - **B2:** Extiende etiquetas de confidencialidad a los recursos del sistema. Se asignan a los dispositivos físicos, para seleccionar los usuarios que pueden usarlos.
  - **B3:** Permite crear listas de control de acceso que denotan usuarios a los que no se concede acceso a un objeto nombrado dado. La TCB contiene un mecanismo que detecta violaciones de seguridad, avisando al administrador y finalizando los sucesos.
- **A – Nivel alto de protección:** Funciona arquitectónicamente igual que B3, pero contiene técnicas de verificación con más alto grado que las especificadas por TCB.

20.9 Ejemplo de modelo de seguridad: Windows NT

Va desde seguridad mínima (por defecto) hasta C2. El administrador puede configurarlo al nivel deseado mediante C2config.exe. El modelo se basa en el concepto de “cuentas de usuario”. Permitiendo crear cualquier cantidad de cuentas de usuario que se pueden agrupar de cualquier manera.
Cuando un usuario ingresa al sistema, NT crea un *testigo de seguridad* que contiene todos los datos y permisos del usuario, y que va a ser accedido por los procesos cada vez que el usuario quiere acceder a un objeto. La validación de una cuenta de usuario se efectúa a través del nombre de usuario y contraseña.

NT cuenta con una auditoría integrada y permite vigilar amenazas de seguridad comunes. Los atributos de seguridad de un objeto NT se describen con un descriptor de seguridad que contiene el identificador de seguridad del propietario del objeto (quien puede cambiar los permisos), un identificador de seguridad de grupo, una lista de control de acceso discrecional (usuarios o grupos a los que se le permite o prohíbe el acceso) y una lista de control de acceso al sistema (guarda mensajes de auditoria del sistema).
NT clasifica los objetos como contenedores o no contenedores. Los contenedores pueden contener lógicamente otros objetos (Ej: Directorios).
El “subobjeto” hereda los permisos del padre. Si se cambian los del padre, cambian los del hijo.

21 El sistema UNIX

21.1 Historia

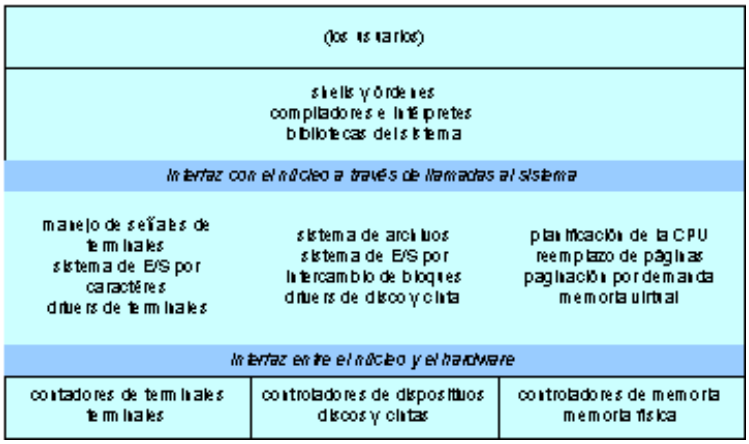
La primera versión fue en 1969 en Bell Laboratories, y MULTICS tuvo una gran influencia sobre él. La tercer versión fue el resultado de escribir la mayor parte del SO en lenguaje C, en lugar de lenguaje ensamblador, agregando funciones de multiprogramación.
La primera versión liberada fue la 6, en 1976, y se utilizó en universidades.
Con el pasar del tiempo se fueron agregando funciones de memoria virtual, paginación y memoria compartida. Se apoyaron los protocolos TCP/IP, logrando gran popularidad y permitiendo un masivo crecimiento de Internet. Se implementó una nueva interfaz con el usuario (C-shell), un nuevo editor de textos (vi), y compiladores.
En 1986 se liberó la versión 4.3BSD, sobre la que se basa este capítulo.
El conjunto actual de sistemas UNIX no está limitado por los producidos por Bell Laboratories, por lo tanto también existen una gran cantidad de interfaces de usuario que los distintos proveedores distribuyen.
UNIX está disponible para casi todos los computadores de propósito general, como PC’s, estaciones de trabajo, etc. en entornos que van de académicos a militares.
Se trata de crear un standard para lograr una única interfaz y popularizar aún mas el sistema.

21.2 Principios de diseño

UNIX se diseñó como un sistema de tiempo compartido. El shell es sencillo y puede ser sustituido por otro si se desea. El sistema de archivos es un árbol con múltiples niveles, y las dependencias y peculiaridades de los dispositivos se mantienen en el núcleo hasta donde es posible, confinadas en su mayor parte a los drivers de dispositivos.
UNIX apoya múltiples procesos. La planificación de la CPU es un sencillo algoritmo de prioridades. Se usa paginación por demanda para gestión de memoria, usando un área de SWAP si hay hiperpaginación.
El sistema es sumamente flexible, y cuenta con recursos para ampliarlo fácilmente. Al ser diseñado para programadores, cuenta con recursos como *make* (útil para compilar archivos fuentes de un programa) y el *Sistema de control de código fuente* (SCCS) para mantener versiones sucesivas de archivos sin necesidad de almacenarlos. Está escrito en C, por lo tanto es portable.

21.3 Interfaz con el programador

Consta de dos partes separables, el núcleo y los programas de sistemas.
Está formado por capas, y los servicios del SO son proporcionados a través de llamadas al sistema. Las llamadas al sistema definen la *interfaz con el programador*, y el conjunto de programas del sistema, la *interfaz con el usuario*, las dos apoyadas por el núcleo.
Las llamadas al sistema se agrupan en tres categorías explicadas a continuación.



21.3.1 Manipulación de archivos

Un *archivo* en UNIX es una secuencia de bytes. El núcleo no impone ninguna estructura de archivo, aunque los diferentes programas esperan distintos niveles de estructura.
Están organizados en *directorios* con estructura de árbol. Los directorios también son archivos con información para encontrar otros archivos. Un *nombre de camino* (path) consiste en nombres de archivo (directorios) separados por '/', donde la base de partida es el directorio root.

UNIX tiene:

- **Nombres de camino absoluto:** Parten de la raíz del sistema y se distinguen por comenzar con una diagonal. Ej. "/usr/local".
- **Nombres de camino relativo:** Parten del directorio actual.

Maneja enlaces:

- **Simbólicos o blandos:** Son archivos que contienen el nombre de camino de otro archivo. Pueden apuntar a directorios y cruzar fronteras de sistemas de archivo (a diferencia de los duros).
- **Duros:**
  - El nombre del archivo '.' es un enlace duro al directorio mismo.
  - El nombre del archivo '..' es un enlace duro al directorio padre.

Los dispositivos de hardware tienen nombres en el sistema de archivos. Los directorios comunes son:

- **/bin:** Contiene binarios de los programas UNIX.
- **/lib:** Contiene bibliotecas.
- **/user:** Contiene los directorios de usuarios.
- **/tmp:** Contiene archivos temporales.
- **/etc.**: Contiene programas administrativos.

Para trabajar con archivos y directorios uso las siguientes llamadas al sistema:

- **creat:** Crea un archivo:
- **open, close:** Abre y cierra archivos.
- **read, write:** Trabaja sobre el archivo.
- **link, unlink:** Crea o destruye enlaces duros a archivos.
- **symlink:** Crea un enlace simbólico al archivo.
- **mkdir, rmdir:** Crea o elimina directorios.
- **cd:** Cambia de directorio.

21.3.2 Control de procesos

Los procesos se identifican con su *identificador de proceso* (PID), y se crea un proceso nuevo con la llamada al sistema **fork**. El proceso nuevo consiste en una copia del espacio de direcciones del padre, y ambos continúan su ejecución en la instrucción siguiente al **fork**, con la diferencia que el código de retorno del **fork** es 0 para el hijo, y es el PID del hijo para el padre.
El proceso termina con una llamada al sistema **exit**, y el padre puede esperarlo con **wait**. El hijo al terminar queda como *difunto* por si el padre necesita su información. En el caso que el padre termine, sin esperar por el hijo, el hijo pasa a ser un proceso *zombie*. Todos los procesos de usuario son hijos de un proceso original llamado *init*.
Tipos de identificador de usuario:

- **Efectivo:** Se usa para los permisos de acceso a archivos, y se hace igual al id del usuario dueño del archivo, para abrirlo.
- **Real:** Es el id del usuario.

Esto permite a ciertos usuarios tener privilegios más allá de los ordinarios, pero controlados por el SO.

21.3.3 Señales

Son un recurso para manejar excepciones, y son similares a las interrupciones de software.

- **SIGINT:** Interrupción. Detiene una orden antes de que termine de ejecutarse. Se produce con el carácter ^C.
- **SIGQUIT:** Abandonar. Detiene un programa y vacía su memoria en un archivo llamado core (^bs).
- **SIGILL:** Instrucción no válida.
- **SIGSEV:** Acceso inválido a memoria.
- **SIGKILL:** No se puede pasar por alto. Sirve para terminar un proceso desbocado que no hace caso a otras señales como SIGINT o SIGQUIT.

No hay prioridades entre señales, y si mando dos del mismo tipo a un proceso, se sobrescriben. También existen señales para entregar datos urgentes en una red, o dormir procesos hasta que sean interrumpidos.

21.3.4 Grupos de procesos

Grupos relacionados de procesos podrían cooperar para realizar una tarea común, y es posible enviar señales a todos los procesos del grupo.
Solo un grupo de procesos puede usar una terminal de E/S en un momento dado. Este trabajo de *primer plano* tiene la atención del usuario de la terminal, mientras los trabajos de *segundo plano* realizan otras tareas.

21.3.5 Manipulación de información

Existen llamadas al sistema para devolver y ajustar temporalizadores, hora actual, y para obtener el PID de un proceso (getpid), nombre de una máquina (gethostname), etc.

21.3.6 Rutinas de biblioteca

La interfaz de llamadas al sistema de UNIX se apoya con bibliotecas y archivos de cabecera. Estos últimos proporcionan estructuras de datos complejas que se emplean en las llamadas. Por ejemplo <stdio.h> proporciona una interfaz que lee y escribe miles de bytes usando buffers locales. Hay otras para funciones matemáticas, acceso a redes, etc.

21.4 Interfaz con el usuario

Los programadores y usuarios manejan los programas de sistema que UNIX provee. Estos efectúan llamadas al sistema que en muchos casos no son obvias para el usuario.

- **Programa ls:** Lista los nombres de archivos del directorio actual. Hay opciones:
  - **- l:** Listado largo. Muestra el nombre de archivo, dueño, protección, fechas y tamaño.

- **Programa cp:** Copia un archivo existente.
- **Programa mv:** Cambia un archivo de lugar o nombre.
- **Programa rm:** Borra archivos
- **Programa cat:** Muestra el contenido de un archivo en la pantalla.

También hay editores (vi), compiladores (C, PASCAL), programas para comparar (cmp, diff), buscar patrones (grep), etc.

21.4.1 Shells y órdenes

Los programas normalmente se ejecutan con un intérprete de órdenes, que en el caso de UNIX es un proceso de usuario como cualquier otro llamado shell. Hay diferentes shells, pero todos tienen órdenes en común.
El shell indica que está listo exhibiendo una señal de espera (prompt) que depende del tipo de shell (ej.: en shell C es “%”), y la orden de usuario es de una única línea.
Generalmente el shell espera (wait) hasta que terminó la tarea que el usuario ordenó, pero se pueden ejecutar *tareas en segundo plano* donde la tarea queda ejecutándose y el shell sigue interpretando órdenes.
El shell C cuenta con un *control de trabajos* que permite transferir procesos entre el primer y el segundo plano.

21.4.2 E/S estándar

Los procesos pueden abrir archivos a discreción, pero siempre tienen abiertos tres descriptores de archivo en su inicio:

- **0:** Entrada estándar.
- **1:** Salida estándar.
- **2:** Error estándar.

Y éstos, les permite trabajar con ellos (ej: pantalla, teclado, etc.).
Esos archivos son por defecto, pero puedo asignar otros si es necesario (ej: impresora como salida). Este cambio es llamado *redirección de E/S.*

21.4.3 Conductos, filtros y guiones de shell

Se usa un conducto para llevar datos de un proceso a otro. En UNIX se usa una barra vertical ‘|’ (pipe), como por ejemplo “ls | lpr” que imprime lo que devolvió el comando ls.
Una orden que pasa su entrada estándar a su salida estándar realizando un proceso sobre ella, es llamada *filtro*.
Un archivo que contiene órdenes de shell es un *guión de shell* (shell script). Se ejecuta como cualquier orden y se invoca el shell apropiado para leerlo.

21.5 Gestión de procesos

ES fácil crear y manipular procesos en UNIX. Los procesos se representan con bloques de control, y se almacenan en el núcleo, el cual los usa para controlar el proceso y planificar la CPU.

21.5.1 Bloques de control de procesos

La *estructura de proceso* contiene lo que el sistema necesita para saber acerca de él. Existe un arreglo de estructuras de proceso, cuyo tamaño se define al enlazar el sistema. El planificador implementa la cola de procesos listos como una lista doblemente enlazada de estructuras e proceso.
El *espacio de direcciones virtual* del proceso se divide en segmentos de texto (código del programa), datos y pila. Los dos últimos están en el mismo espacio de direcciones, pero como son variables crecen en direcciones opuestas.
Las *tablas de páginas* registran información de la correspondencia entre memoria virtual y física.
En la *estructura de usuario* se guardan los id de usuario y grupo, el directorio actual, la tabla de archivos abiertos, etc.
Cuando un proceso se ejecuta en modo de sistema, se utiliza una pila de núcleo para el proceso, en lugar de la pila del proceso.
La pila de núcleo y la estructura de usuario constituyen el *segmento de datos del sistema* de ese proceso.

El **fork** copia los datos y la pila en el nuevo proceso, al contrario del **vfork**.

21.5.2 Planificación de CPU

Está diseñada para beneficiar los procesos interactivos. Cada proceso tiene una *prioridad de planificación* (a mayor número, menor prioridad). Los procesos que realizan tareas importantes tienen prioridades menores que “pzero” y no se pueden matar. Los demás tienen prioridades positivas.
Cuanto más tiempo de CPU acumula un proceso, más baja su prioridad, y se usa envejecimiento para evitar inanición.
En el núcleo, ningún proceso desaloja a otro.

21.6 Gestión de memoria

21.6.1 Intercambio

Un proceso más grande que la memoria no podía ser ejecutado, y los segmentos de datos de sistema y usuario, que se mantenían en memoria principal, causaban fragmentación externa.
La asignación de memoria virtual y espacio de intercambio es por primer ajuste. Si la memoria aumentaba se buscaba un nuevo hueco y se copiaba. Si no existía tal hueco, se cambiaba a disco.
El *proceso planificador* decide que procesos se deben intercambiar, y se despierta cada 4 segundos para revisar. Para evitar hiperpaginación, no se intercambia a disco un proceso que no haya estado un cierto tiempo en memoria.
Todas las particiones de swap deben tener el mismo tamaño (potencia de dos) y deben poder accederse con diferentes brazos del disco.

21.6.2 Paginación

Utiliza paginación por demanda, eliminando la fragmentación externa, y reduciendo la cantidad de intercambios.
Algunas arquitecturas, no cuentan con un bit de referencia a página en hardware, impidiendo el uso de muchos algoritmos de gestión de memoria. Utiliza una modificación del algoritmo de segunda oportunidad. La escritura de páginas se efectúa en cúmulos para mejorar el desempeño.
Se realizan verificaciones para que el número de páginas en memoria para cada proceso no se reduzca demasiado, o no aumente demasiado. El encargado es *pagedaemon* que está dormido hasta que el número de marcos libres se reduce más allá de un umbral.
En VAX el tamaño de página es pequeño causando ineficiencias de la E/S, así que se juntan en grupos de a dos. El tamaño de página efectivo no necesariamente es idéntico al tamaño de página de hardware, aunque si debe ser múltiplo.

21.7 Sistema de archivos

21.7.1 Bloques y fragmentos

El sistema de archivos está ocupado por *bloques de datos* los cuales deben ser mayores a 512 bites, por razones de velocidad, pero no muy grandes por causa de la fragmentación interna.
UNIX usa dos tamaños de bloque. Todos los bloques de un archivo son grandes menos el último que es llamado *fragment*. Los tamaños se establecen al crear el sistema de archivos, obligando que la proporción bloque-fragmento sea 8:1.
Se debe conocer el tamaño del archivo antes de copiarlo a disco, a fin de saber si debo copiar un bloque o un fragmento. De otra forma tendría que copiar 7 fragmentos, y luego pasarlos a un bloque, causando ineficiencia.

21.7.2 I-nodos

Un archivo se representa con un *i-nodo* que es un registro que guarda información acerca del archivo. Contiene:

- Identificadores de usuario y grupo.
- Hora de última modificación y último acceso.

- Cuenta del nnúmero de enlaces duros.
- Tipo de archivo.
- 15 punteros a bloques
  - **12 primeros:** Apuntan a *bloques directos*, o sea son una dirección de un bloque de datos.
  - **3 siguientes:** Apuntan a *bloques indirectos*.
    - **Primero:** Apunta a un *bloque de indirección simple*, que es un bloque índice con punteros a bloques de datos.
    - **Segundo:** Es un puntero a un *bloque de indirección doble*, el cual apunta a un bloque índice que contiene direcciones de bloques índices que apuntan a bloques de datos.
    - **Tercero:** Es un puntero a un *bloque de indirección triple*. En UNIX no se usa, ya que el tamaño de los archivos puede alcanzar solo 232 bytes (por ser un sistema de direccionamiento de 32 bits), y para ello alcanza con el segundo.

21.7.3 Directorios

Se representan igual que los archivos, pero difieren en el campo “tipo” del i-nodo, además que el directorio tiene una estructura específica.
Los primeros nombres de cada directorio son ‘.’ y ‘..’. Para buscar un archivo, voy recorriendo paso a paso el camino, y no todo junto, ya que en muchos casos, cuando cambio de dispositivo, también cambia el sistema de archivos. El último nombre de un camino, debe ser el de un archivo.
Debido a la existencia de enlaces, para evitar ciclos, en una búsqueda no puedo pasar por más de 8 enlaces.
Los archivos que no están en disco (dispositivos) no tienen bloques asignados. El núcleo los detecta y llama a los controladores de E/S apropiados.

21.7.4 Transformación de un descriptor de archivo en un i-nodo

Para trabajar con un archivo, debo pasar un descriptor de él como argumento a las llamadas al sistema. El núcleo lo usa como entrada de una tabla que apunta al i-nodo del archivo. La *tabla de archivos abiertos* tiene una longitud fija, por lo tanto no puedo abrir archivos ilimitadamente.

21.7.5 Estructuras de disco

Los dispositivos físicos se dividen en dispositivos lógicos, y cada uno de estos últimos define un sistema de archivos físico, para diferentes usos. Una de las particiones es la de swap.
El primer sector del dispositivo lógico es el *bloque de arranque* que puede contener un programa de autoarranque primario, que llama al secundario que está en los próximos 7,5 Kb.
21.7.7 Organización y políticas de asignación

El núcleo usa un par *<número de dispositivo lógico, número de i-nodo>* para identificar un archivo. No puedo asegurar la confiabilidad del sistema de archivos, ya que se mantiene en memoria, escribiéndolo a disco cada 30 segundos. Por lo que un corte de energía podría producir inconsistencia.
Se introdujo el sistema de *grupo de cilindros* para localizar bloques de archivos logrando casi no tener que mover la cabeza del disco. Cada cilindro guarda información útil para recuperación. Siempre se trata que todos los bloques de un archivo estén dentro del mismo grupo de cilindros.

21.8 Sistemas de E/S

Se trata de ocultar las peculiaridades de los dispositivos, y se hace en el núcleo con el *sistema de E/S*, que consiste en un sistema de buffers en caché, y drivers para los dispositivos (que conoce las peculiaridades de ellos).
Hay tres tipos principales de dispositivos:

- **Por bloques:** Discos y cintas. Se pueden direccionar directamente en bloques de tamaño fijo. Las transferencias se hacen

a través del *caché de buffers de bloques*.
- **Por caracteres:** Terminales, impresoras, etc. Agrupa los dispositivos que no usan caché de buffers de bloques, como por ejemplo las interfaces de gráficos de alta velocidad.
- **Interfaz de socket**

Un dispositivo se distingue por:

- **Tipo**
- **Número de dispositivo:** Se divide en:
  - **Número de dispositivo principal:** Indica el lugar en el arreglo de dispositivo, para encontrar puntos de ingreso al driver apropiado.
  - **Número de dispositivo secundario:** Es interpretado por el driver, como por ejemplo una partición lógica de disco, etc.

21.8.1 Caché de buffers de bloques

Consiste en varias cabeceras de buffer que pueden apuntar a memoria física, a un número de dispositivo, o a un número de bloque en el dispositivo.
Los buffers no usados se mantienen en tres listas:

- **Recientemente usados:** Enlazados en orden LRU (Lista LRU).
- **No usados recientemente:** Sin contenido válido (Lista AGE).
- **Vacíos:** No tienen memoria física asociada (Lista EMPTY).

Al requerir una lectura, se busca en el caché de buffers. Si se encuentra, se usa, y si no se encuentra, se lee del dispositivo, y se guardan los datos en el caché, eligiendo un buffer de la lista AGE, o si está vacía, de la lista LRU.
La cantidad de datos del buffer es variable, y el tamaño mínimo es el tamaño del cúmulo de paginación. El tamaño del caché, tiene un efecto considerable en el desempeño del sistema, dependiendo de él, la cantidad de transferencias de E/S que se deben realizar.
Cuando escribo en una cinta, debo sincronizar los buffers, para que escriban en un orden determinado. Cuando escribo en un disco, los puedo sincronizar para lograr el menor movimiento posible de la cabeza del disco.

21.8.2 Interfaces con dispositivos crudas

Se les llama a las interfaces por caracteres que tienen los dispositivos por bloques, y lo que hacen
es pasar por alto el caché de bloques. En estos casos, el tamaño de la transferencia está limitado por los dispositivos físicos, algunos de los cuales requieren un número par de bytes.

21.8.3 Listas C

Los controladores de terminales usan un sistema de buffer de caracteres, que mantiene bloques pequeños de caracteres en listas enlazadas. Tanto las salidas como entradas se controlan con interrupciones.
Los editores de pantalla completa y otros programas que necesitan reaccionar con cada digitación, usan el *modo crudo* que pasa por alto el buffer de caracteres.

21.9 Comunicación entre procesos (IPC)

La comunicación entre procesos no es uno de los puntos fuertes de UNIX. En muchos casos no se permite la *memoria compartida* porque algunas arquitecturas no apoyaban su uso. En los actuales si, pero de todas formas, presenta problemas en un entorno de red, ya que los accesos por la red no son tan rápidos como los accesos locales, perdiendo la ventaja de velocidad en la memoria compartida.

21.9.1 Sockets

Es el mecanismo de IPC característico de UNIX. Un *conducto* permite establecer un flujo de bytes unidireccional entre dos procesos, y se implementan como archivos ordinarios (con algunas excepciones). El tamaño es fijo, y si esta lleno, los procesos que intentan escribir en él, se suspenden. Como ventaja, es pequeño, y su escritura no pasa por disco, sino a través de los buffers de bloques.
En 4.3BSD se implementan con sockets, que proporcionan una interfaz general, tanto en la misma máquina, como en redes. Puede ser usado entre procesos no relacionados entre si.
Los socket tienen vinculadas *direcciones*, cuya naturaleza depende del *dominio de comunicación del socket.* Dos procesos que emplean el mismo dominio, tienen el mismo formato de direcciones. Los tres dominios que se implementan son el dominio de UNIX (AF_UNIX), cuyo formato de direcciones es el de los nombres de caminos del sistema de archivos (ej: /casa/auto), el dominio de Internet (AF_INET) y el de Servicios de red (AF_NS).
Hay varios tipos de sockets, los cuales se clasifican dependiendo del servicio que realizan, y el recurso socket tiene un conjunto de llamadas específicas al sistema. Luego de establecer una conexión por sockets, se conocen las direcciones de ambos puntos, y no se necesita ningún tipo de información para transferir datos. Algunos tipos de socket, no conocen conexiones (datagramas), y se les debe especificar el destinatario del mensaje.

21.9.2 Soporte para redes

Reconoce gran cantidad de protocolos de Internet, y el núcleo se diseñó con miras de facilitar la implementación de protocolos adicionales.
Los procesos de usuario se comunican con los protocolos de red a través del recurso de sockets; y los sockets se apoyan con protocolos, los cuales prestan sus servicios.
Suele haber un *driver de interfaz de red* para cada tipo de controlador de red, que maneja las características específicas de la red local, asegurando que los protocolos no tengan que preocuparse por ello. Las funciones de la interfaz de red, dependen del hardware de red, y de las funciones que él apoya (como trasmisión segura, etc.).

22 El sistema Linux

22.1 Historia

Es muy parecido a UNIX y tiene la compatibilidad fue una de las metas de su diseño. Su desarrollo se inició en 1991, para el 80386 (de 32 bits).
El código fuente se ofrecía gratuitamente en Internet, por lo tanto su historia ha sido una colaboración de muchos usuarios de todo el mundo.
Se divide entre el *núcleo* y un *sistema* Linux. El núcleo fue desarrollado de cero por la comunidad Linux, y el sistema incluye gran cantidad de componentes. Es un entorno estándar para aplicaciones y programación, y una *distribución* Linux incluye todos los componentes estándar, junto con un conjunto de herramientas administrativas que simplifican la instalación y desinstalación de paquetes, etc.

22.1.1 El núcleo de Linux

La primer versión, en 1991 no trabajaba con redes y sólo se ejecutaba con procesadores compatibles con el 80386, además, el soporte de drivers era limitado. No tenía soporte para archivos con correspondencia en memoria virtual, pero manejaba páginas compartidas. Solo reconocía un único sistema de archivos (Minix), pero implementaba correctamente procesos UNIX.
En 1994, salió la siguiente versión que apoyaba el trabajo con redes, con los protocolos TCP/IP y UNIX, sockets, y nuevos drivers de dispositivos (para redes o Internet). También incluía nuevos sistemas de archivos y reconocía controladores SCSI. El subsistema de memoria virtual se extendió para apoyar el intercambio por paginación, pero de todas formas seguía restringido a la plataforma Intel. Además se implementó la comunicación entre procesos (IPC) de UNIX, con memoria compartida,

semáforos y cola de mensajes.
En 1994 salió la versión 1.2, soportando nuevo hardware, y la arquitectura de bus PCI. Se añadió soporte para emular el sistema DOS. Incluía soporte para otros procesadores, aunque no completo.
En 1996 la versión 2.0 Soportaba múltiples arquitecturas (inclusive la Alpha de 64 bits) y arquitecturas multiprocesador. Se mejoró la gestión de memoria, y se introdujo soporte de hilos internos del núcleo y carga automática de módulos por demanda.

22.1.2 El sistema Linux

El sistema Linux es código común a varios sistemas operativos tipo UNIX. Las bibliotecas principales se comenzaron como GNU, pero se fueron mejorando y agregando funcionalidades. El sistema Linux es mantenido por una red de desarrolladores que colaboran por Internet, pero se tiene un libro de normas, para asegurar la compatibilidad.

22.1.3 Distribuciones de Linux

Incluyen el sistema Linux básico, utilitarios de instalación y administración, paquetes, herramientas, procesadores de texto, editores, juegos, etc. facilitando la tarea de la persona que lo quiere instalar en su máquina, y poniendo cada componente en su lugar apropiado. Hay distribuciones comerciales y no comerciales como Red Hat, SuSE, Mandrake, etc.

22.1.4 Licencias de Linux

El núcleo se distribuye bajo la Licencia Publica General de GNU. Los autores renunciaron a los derechos de autor sobre el software, pero los derechos de código siguen siendo propiedad de dichos autores. Linux es software gratuito, y nadie que haga un derivado, lo puede registrar como propio.

22.2 Principios de Diseño

Linux es un sistema multiusuario, multitarea con un conjunto completo de herramientas compatibles con UNIX. Linux desde sus principios procuró sacar el máximo de funcionalidad posible en recursos limitados, por lo tanto puede ejecutarse en muchos sistemas pequeños.

22.2.1 Componentes de un sistema Linux

El sistema Linux se divide en tres partes principales de código:

- **Núcleo:** Mantiene las abstracciones importantes del sistema operativo como memoria virtual y procesos. Implementa todas las operaciones necesarias para calificar como SO.
- **Bibliotecas del sistema:** Definen un conjunto estándar de funciones a través de las cuales las aplicaciones pueden interactuar con el núcleo y que implementan gran parte de la funcionalidad del SO, que no necesita todos los privilegios del código de núcleo.
- **Utilitarios del sistema:** Son programas que realizan tareas de administración. Implementa por ejemplo los “demonios” que son procesos que se ejecutan de forma permanente.

El código de núcleo se ejecuta en el modo privilegiado del procesador (*modo de núcleo*) con pleno acceso a todos los recursos físicos. No se incorpora código del modo usuario en el núcleo, el cual es colocado en bibliotecas del sistema.
El núcleo de Linux se crea como algo separado, aislado (estructura monolítica) al igual que UNIX. Este sistema tiene muchas ventajas, dado que el núcleo se mantiene en un solo espacio de direcciones, no se requieren conmutaciones de contexto cuando un proceso invoca a una función del SO. Además dentro de ese espacio de direcciones se encuentra el código de planificación, memoria virtual central, todo el código de núcleo, incluido el de los drivers de dispositivos, sistema de archivos y trabajo con redes.
El sistema operativo que el núcleo de Linux ofrece no se parece en absoluto a un sistema UNIX. Faltan muchas características extra de UNIX, y las funciones que si se ofrecen no son necesariamente del formato que utiliza UNIX. El núcleo no mantiene una interfaz con el SO que ven las aplicaciones, mas bien las aplicaciones para comunicarse con el SO deben hacer llamadas a

las bibliotecas del sistema las cuales solicitan servicios del SO.
Las bibliotecas del sistema ofrecen funcionalidad, permitiendo hacer solicitudes de servicio al núcleo, y otras rutinas como funciones matemáticas, etc.
Los programas en modo usuario, incluyen programas necesarios para iniciar el sistema, configurar la red, programas de inicio de sesión de usuarios, etc. También incluye operaciones sencillas como listar directorios, trabajar con archivos, y otras más complejas como procesadores de texto, etc.

22.3 Módulos del Núcleo

El núcleo de Linux tiene la facultad de cargar y descargar módulos del núcleo cuando se le pide hacerlo. Estos se ejecutan en modo de núcleo privilegiado, teniendo acceso a todas las capacidades de hardware. Los módulos pueden implementar desde un driver hasta un protocolo de redes.
La ventaja de esta modularización, es que cualquiera puede variar a su gusto algunos de los módulos núcleo (usualmente no es necesario re-compilar todo el núcleo, después que se hizo una variación en algún modulo del sistema). Otra ventaja es que se puede configurar el sistema con un núcleo mínimo, y agregar controladores mientras se usa, como por ejemplo el montaje de un CD-ROM, etc.

El soporte de módulos en Linux tiene tres componentes principales:

- **Gestión de módulos:** Permite cargar módulos en la memoria y comunicarse con el resto del sistema.
- **Registro de controladores:** Permite a los módulos notificar al resto del núcleo que está disponible un nuevo controlador.
- **Resolución de conflictos:** Permite a los drivers reservar recursos de hardware y protegerlos.

22.3.1 Gestión de módulos

La carga de un módulo no es solo la carga en memoria del núcleo, sino que el sistema debe asegurarse de que cualquiera referencia que el módulo haga a símbolos o puntos de ingreso del núcleo se actualicen de modo que apunten a las posiciones correctas dentro del espacio de direccionamiento del núcleo. Si el sistema no puede resolver alguna referencia del módulo, el módulo se rechaza.
La carga se efectúa en dos etapas:

- Se reserva un área continua de memoria virtual del núcleo para el módulo, y se devuelve un puntero a esa dirección.
- Se pasa el módulo junto con la tabla de símbolos al núcleo, copiando el módulo sin alteraciones en el espacio previamente asignado.

Otro componente de la gestión de módulos encontramos el solicitador de módulos, que es una interfaz de comunicación con la cual puede conectarse un programa de gestión de módulos. La función que cumple es de avisar cuando el proceso necesite un driver, o sistema de archivo etc., que no este cargado actualmente, y dará la oportunidad de cargar ese servicio.

22.3.2 Registro de controladores

El núcleo mantiene tablas dinámicas de todos los controladores conocidos, y proporciona un conjunto de rutinas que permite añadir o quitar controladores de de estas tablas en cualquier momento. Las tablas de registro incluyen los siguientes elementos:

- **Drivers de dispositivos:** Incluyen los dispositivos de caracteres (impresoras, teclados), dispositivos por bloques (Disco) y dispositivos de interfaz de red.
- **Sistemas de archivos:** Puede ser cualquier cosa que implemente rutinas de invocación del sistema de archivos virtual de Linux.
- **Protocolos de red:** Puede implementar un protocolo de red completo o un conjunto de reglas de un firewall.
- **Formato binario:** Especifica una forma de reconocer, y cargar, un nuevo tipo de archivo ejecutable.

22.3.3 Resolución de conflictos

Linux ofrece un mecanismo central de resolución de conflictos que ayuda a arbitrar el acceso a ciertos recursos de hardware. Sus objetivos son:

- Evitar que los módulos entren en conflictos al tratar de acceder a recursos de hardware.
- Evitar que los sondeos de drivers que auto detectan la configuración de un dispositivo interfieran con los drivers de dispositivos existentes.
- Resolver conflictos entre múltiples drivers que tratan de acceder al mismo hardware.

Para esto el núcleo mantiene una lista de recursos de hardware asignados. Por lo que si cualquier controlador de dispositivo quiere acceder a un recurso, debe reservar primero el recurso en la lista del núcleo. Si la reservación se rechaza, debido a que esta en uso, corresponde al módulo decidir como proceder.

22.4 Gestión de procesos

22.4.1 El modelo de proceso fork/exec

A diferencia de UNIX, en Linux cualquier programa puede invocar **execve** en cualquier momento sin necesidad de crear un proceso nuevo. Bajo Linux la información que contiene un proceso se puede dividir en varias secciones específicas, descriptas a continuación.

22.4.1.1 Identidad del proceso

Consiste en los siguientes elementos:

- **Identificador del proceso (PID):** Es único y sirve para especificar procesos al SO. También puede tener identificadores adicionales de grupo.
- **Credenciales:** Cada proceso debe tener asociado un identificador de usuario y uno o mas identificadores de grupo, que determinan los derechos de acceso a lo recursos y archivos del sistema que el proceso tiene.
- **Personalidad:** Cada proceso tiene asociado un identificador de personalidad el cual sirve para modificar un poco las semánticas de ciertas llamadas al sistema. Sirven para crear compatibilidad con algunos sistemas UNIX.

22.4.1.2 Entorno del proceso

Se hereda del padre y consiste en dos vectores terminados con el carácter nulo:

- **Vector de argumentos:** Lista los argumentos de línea de órdenes que se usan para invocar el programa que se esta ejecutando (por convención comienza con el nombre del programa del mismo).
- **Vector de entorno:** Es una lista de pares “NOMBRE = VALOR” que asocia variables de entorno con nombre a valores de texto arbitrarios.

22.4.1.3 Contexto de un proceso

La identidad del proceso no se cambia en su ejecución. El contexto es el estado del programa en ejecución en cualquier instante dado, y cambia constantemente. Se divide en:

- **Contexto de planificación:** Contiene la información que el planificador necesita para suspender y reiniciar el proceso. Se incluye copias guardadas de todos los registros del proceso. También incluye información acerca de la prioridad de planificación, y de señales pendientes que esperan ser entregadas al proceso.

- **Contabilidad:** El núcleo mantiene información acerca de los recursos que cada proceso está consumiendo, y el total de recursos que el proceso ha consumido durante toda su existencia.
- **Tablas de archivos:** Es un arreglo de punteros a estructuras de archivos del núcleo. Al efectuarse una llamada al sistema para E/S con archivos, los procesos se refieren a los archivos empleando su índice en esta tabla.
- **Contexto del sistema de archivos:** Guarda los nombres de los directorios raíz y por omisión que se usarán para buscar archivos.
- **Tabla de manejadores de señales:** Define la rutina invocara cuando lleguen señales específicas.
- **Contexto de memoria virtual:** Describe todo el contenido del espacio de direcciones privado de dicho proceso.

22.4.2 Procesos e Hilos

Dos hilos adentro de un proceso, a diferencia de los procesos, comparten el mismo espacio de direcciones.
El núcleo de Linux maneja de forma sencilla la creación de hilos y procesos, ya que utiliza exactamente la misma representación interna para todos. Un hilo no es más que un proceso nuevo que comparte el mismo espacio de direcciones que su padre. La distinción entre un proceso y un hilo se hace sólo cuando se crea un hilo nuevo con la llamada “clone”. Mientras que “fork” crea un proceso nuevo que tiene su propio contexto totalmente nuevo, “clone” crea un proceso nuevo que tiene identidad propia pero que puede compartir las estructuras de datos de su padre.
La llamada al sistema “fork” es un caso especial de la llamada “clone” que copia todos los subcontextos y no comparte ninguno.

22.5 Planificación

En Linux la planificación no cumple solo con la función de ejecución e interrupción de procesos, sino que también tiene asociado la ejecución de diversas tareas del núcleo.

22.5.1 Sincronización del núcleo

La sincronización del núcleo implica mucho más que la simple planificación de procesos. Se requiere una estructura que permita la ejecución de las secciones críticas del núcleo sin que otra sección crítica la interrumpa, para que se puedan ejecutar concurrentemente las tareas del núcleo con las rutinas de servicio de interrupciones.

La primera técnica para solucionar el problema radica en hacer no expropiable el código de núcleo normal. Una vez que un fragmento del código del núcleo comienza a ejecutarse se puede garantizar que será el único código del núcleo que se ejecute hasta que pase algo de lo siguiente:

- **Interrupción:** Solo son un problema si sus rutinas de servicio contienen secciones críticas.
- **Fallo de página:** Si una rutina del núcleo trata de leer o escribir la memoria del usuario, esto podría incurrir en un fallo de página, que requiera E/S de disco y el proceso en ejecución se suspenderá hasta que termine la E/S.
- **Llamada al planificador desde el código del núcleo:** El proceso se suspenderá y habrá una replanificación.

En conclusión con esta técnica el código del núcleo nunca será desalojado por otro proceso, y no es necesario tomar medidas especiales para proteger las secciones críticas, teniendo como precondición que las secciones críticas no tenga referencias a memoria de usuario ni esperas para que termine una E/S.

La segunda técnica que Linux utiliza para proteger a las secciones críticas es inhabilitando las interrupciones durante una sección critica, garantizando que podrá continuar sin que haya peligro de acceso concurrente a estructuras de datos compartidas. El problema de este sistema es que muy costoso.

Linux plantea una solución intermedia para poder aplicar secciones críticas largas sin tener que inhabilitar las interrupciones.

Para esto divide en varias partes el servicio de interrupciones.

Manejador de interrupciones de la mitad superior
Manejador de interrupciones de mitad inferior
Rutinas de servicios del núcleo – sistema (no desalojables)
Programas en modo usuario (desalojables)

Cada nivel puede ser interrumpido por código que se ejecute en un nivel más alto, pero nunca por código del mismo nivel, o de un nivel inferior, con excepción del código en modo de usuario.

22.5.2 Planificación de procesos

Linux tiene dos algoritmos de planificación de procesos independientes:

- **Algoritmo de tiempo compartido para una planificación expropiativa justa entre procesos:** Es implementado con prioridad *basada en créditos*. Cada proceso tiene un cierto número de créditos de planificación, cuando es preciso escoger una nueva tarea para ejecutar se selecciona el proceso con mas créditos. Cada vez que se produce una interrupción del temporizador el proceso pierde un crédito, cuando llegan a cero se suspende y se escoge otro proceso. Además se tiene una forma de renovación de créditos en las que se añaden créditos a todos los procesos del sistema (crédito = crédito /2 + prioridad).
- **Algoritmo para tareas en tiempo real en las que las prioridades absolutas son más importantes que la equitatividad:** Con este sistema el procesa que entra primero es el proceso que tiene mas prioridad (utiliza una cola circular), si hay dos con la misma prioridad ejecuta el proceso que ha esperado mas tiempo. Este planificador es de tiempo real blando, ofrece garantías acerca de las prioridades pero no acerca de la rapidez.

22.5.3 Multiprocesamiento simétrico

El núcleo de Linux implementa correctamente el *multiprocesamiento simétrico* (SMP). La desventaja que tiene, es que procesos e hilos se pueden ejecutar en paralelo en distintos procesadores, pero no puede estar ejecutándose código del núcleo en más de un procesador.

22.6 Gestión de memoria

La gestión de memoria en Linux tiene dos componentes:

- **Sistema de gestión de memoria física:** Se encarga de asignar y liberar páginas, grupos de páginas y bloques pequeños de memoria.
- **Sistema de gestión de memoria virtual:** Maneja la memoria virtual, que es memoria que tiene una correspondencia con el espacio de direcciones de procesos en ejecución.

22.6.1 Gestión de memoria física

El administrador primario de memoria física es el asignador de páginas, encargado de asignar y liberar todas las páginas físicas además de asignar intervalos de páginas contiguas físicamente si se lo solicitan. El asignador utiliza el algoritmo de *montículo de compañeras* para saber las páginas que están disponibles. Si dos regiones compañeras asignadas quedan libres se combinan para formar una región más grande. Si se solicita una pequeña, y no hay, se subdivide recursivamente una grande, hasta lograr el tamaño deseado.
Las funciones del núcleo no tienen que usar el asignador básico para reservar memoria, existen otros subsistemas de gestión de memoria especializados, dentro de ellos encontramos:

- **Memoria Virtual.**
- **Asignador de longitud variable:** Utiliza el asignador de páginas *kmalloc*, el cual cumple solicitudes de tamaño arbitrario que no necesariamente se conocen con anticipación. Este asignador esta protegido contra interrupciones.
- **El caché de buffer:** Es el caché principal del núcleo para dispositivos orientados a bloques.
- **El caché de páginas:** Almacena páginas enteras del contenido de archivos, de todo tipo de dispositivos (no solo bloques).

Los tres sistemas interactúan entre ellos.

22.6.2 Memoria virtual

Se encarga de mantener el espacio de direcciones visible para cada proceso. Se crean páginas por solicitud. Mantiene dos vistas del espacio de direcciones:

- **Vista lógica:** Consiste en un conjunto de regiones que no se traslapan. Cada región se describe internamente como una sola estructura que define las propiedades (lectura, escritura, etc.). Las regiones se enlazan en un árbol binario balanceado.
- **Vista física:** Almacena las tablas de páginas de hardware para el proceso. Guardan la ubicación exacta de la página, que puede estar en disco o memoria. Cada dirección de la tabla guarda un puntero a una tabla de funciones que controla el espacio de memoria indicado.

22.6.2.1 Regiones de memoria virtual

Hay varios tipos dependiendo de las propiedades:

- **Almacenamiento auxiliar:** Describe el origen de las páginas de una región. Se puede respaldar en un archivo, o no respaldar (tipo más sencillo de memoria virtual).
- **Reacción a las escrituras:** La correspondencia entre una región y el espacio de direcciones del proceso puede ser *privada* o *compartida*.
  - **Privada:** Se requiere copiar al escribir para que los cambios sean locales al proceso.
  - **Compartida:** Se actualiza para que todos los procesos vean de inmediato la modificación.

22.6.2.2 Tiempo de vida de un espacio de direcciones virtual

Hay dos formas de crear procesos:

- **Exec:** Se crea un espacio de direcciones virtual nuevo.
- **Fork:** Se copia el espacio de direcciones de un proceso existente (el padre). Cuando debo copiar una región de correspondencia privada, las modificaciones que hagan el hijo y el padre son privadas, por lo tanto la página se debe marcar como “solo lectura”, para que se deba realizar una copia antes de modificarla.

22.6.2.3 Intercambio y paginación

Los primeros sistemas UNIX intercambian a disco procesos enteros. Actualmente se intercambian páginas. El sistema de paginación se divide en dos secciones:

- **Algoritmo de política:** Decide que páginas pasar a disco y cuando escribirlas.
- **Mecanismo de paginación:** Transfiere la página a memoria cuando se la necesita.

Linux emplea un mecanismo modificado del algoritmo de segunda oportunidad.

22.6.2.4 Memoria virtual del núcleo

Linux reserva para el núcleo una región constante, que depende de la arquitectura, y las entradas de la tabla de páginas correspondiente a esa zona, se marcan como protegidas. Esta área contiene dos regiones:

- **Estática:** Contiene referencias de tabla de páginas, a cada página de memoria física disponible en el sistema, para traducir fácilmente direcciones físicas en virtuales cuando se ejecuta código del núcleo.

- **Sin propósito específico:** El núcleo en esta zona puede modificar entradas de tabla de páginas para que apunten donde quiera.

22.6.3 Ejecución y carga de programas de usuario

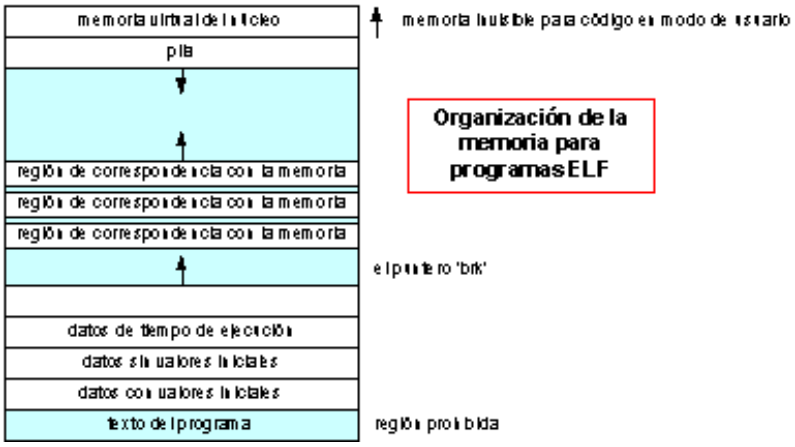
La ejecución se dispara con la llamada al sistema **exec**, que ordena ejecutar un programa nuevo dentro del proceso actual. Las tareas del SO son:

1. Verificar que el proceso invocador tenga permiso de hacerlo.
2. Invocar la rutina de carga para iniciar la ejecución del programa.

No hay una única rutina de carga sino que Linux mantiene una tabla de rutinas cargadoras, ya que en la evolución de Linux, se cambió el formato de los archivos binarios. Antiguamente el formato era .out, actualmente es ELF, que es más flexible y extensible.

22.6.3.1 Correspondencia entre los programas y la memoria

Al comenzar la ejecución, no se carga el programa en memoria física, sino que se establece la correspondencia entre las páginas del archivo binario, y las regiones de memoria virtual. Solo cuando el programa trata de acceder a una página, se genera un fallo de página, y se la trae a la memoria física.
El formato ELF consiste en una cabecera seguida de varias secciones alineadas con páginas. El cargador lee la cabecera y establece correspondencia entre las secciones del programa y la memoria virtual.



22.6.3.2 Enlazado estático y dinámico

Luego de cargar el programa, también se necesita cargar en memoria las funciones del sistema que utiliza este programa. En el caso más sencillo, estás se incrustan en el código binario (enlace estático) con el problema que se repite gran cantidad de código entre todos los programas. El enlace *dinámico* soluciona este problema.
Linux implementa el enlace dinámico a través de una biblioteca. Cuando enlazo el programa, se coloca en él un pequeño programa con instrucciones para cargar esas bibliotecas que necesita.
Las bibliotecas se compilan en código independiente de la posición (PIC) que se puede ejecutar en cualquier dirección de memoria.

22.7 Sistema de archivos

Usa el sistema de archivos estándar de UNIX. Un archivo no tiene que ser un objeto, sino que puede ser cualquier cosa capaz de manejar la entrada y salida de datos, como por ejemplo un driver.
A través de una capa de software, el *sistema de archivos virtual* (VFS) se encarga de abstraer sus peculiaridades.

22.7.1 El sistema de archivos virtual

Está orientado a objetos y tiene dos componentes:

- Definiciones sobre el aspecto del archivo. Los tres tipos principales de objetos son:
  - **i-nodo, objeto archivo:** Son mecanismos empleados para acceder a archivos.
    - **i-nodo:** Representa el archivo como un todo. Mantiene información estándar acerca del archivo.
    - **objeto archivo:** Representa un punto de acceso a los datos del archivo.
  - **Objeto sistema de archivos:** Representa un conjunto conectado de archivos que forma una jerarquía de

directorios, y su principal obligación es dar acceso a los i-nodos.
- Capa de software para manejar dichos objetos.

Se define un conjunto de operaciones que cada uno de los tres tipos debe implementar, así la capa de software, puede realizar funciones sobre los objetos independientemente de su tipo.
Los archivos de directorio en Linux se manejan de una forma un tanto diferente a los demás archivos, por lo tanto tienen definidas operaciones específicas.

22.7.2 El sistema de archivos Linux ext2fs

Es similar al ffs (fast file system) de BSD. Los archivos de directorio se almacenan en disco, al igual que los archivos normales, pero su contenido se interpreta de diferente manera.
Ext2fs no usa fragmentos, sino que realiza todas sus asignaciones en unidades más pequeñas. El tamaño del bloque es de 1kb (aunque también se manejan bloques de 2 y 4 Kb). Está dividido en *grupos de bloques*, y dentro de ellos, el planificador trata de mantener las asignaciones físicamente contiguas, tratando siempre de reducir la fragmentación. Además la fragmentación se compensa con que los bloques están cercanos, por lo tanto no es necesario mover mucho la cabeza del disco para leerlos a todos.

22.7.3 El sistema de archivos proc de Linux

El *sistema de archivos de procesos* es un sistema de archivos cuyo contenido no está almacenado realmente en ninguna parte sino que se calcula bajo demanda, según las solicitudes de E/S de los usuarios.
Cada subdirectorio no corresponde a un directorio de disco, sino a un proceso activo del sistema, y el nombre del directorio es la representación ASCII del PID del proceso.
De esta forma se puede dar un listado de todos los procesos activos con solamente la llamada al sistema **ps**.

22.8 Entrada y salida

Los drivers de dispositivos son archivos, por lo tanto un usuario puede abrir un canal de acceso a un dispositivo de la misma forma que puede abrir cualquier otro archivo. Además, puede utilizar el mismo sistema de protección de archivos para proteger los dispositivos.
Linux divide todos los dispositivos en tres clases:

- **Dispositivos por bloques:** Suelen usarse para almacenar sistemas de archivos, pero también se permite el acceso directo a estos dispositivos para que los programas puedan crear y reparar el sistema de archivos que el dispositivo contiene.
- **Dispositivos por caracteres:** No necesitan apoyar toda la funcionalidad de los archivos normales. Por ejemplo, es útil manejar una búsqueda en un dispositivo de cinta, pero no en un mouse.
- **Dispositivos de red:** Se tratan de manera diferente. Los usuarios no pueden transferir datos directamente, sino que deben abrir una conexión.

22.8.1 Dispositivos por bloques

Para asegurar un rápido acceso a los discos se utiliza el *caché de buffers de bloques* y el *gestor de solicitudes*.

22.8.1.1 El caché de buffers por bloques

El caché de buffers sirve como reserva de buffers para E/S activa y como caché para la E/S finalizada.
El caché de buffers cuenta con dos partes:

- **Los buffers mismos:** Son un conjunto de páginas cuyo tamaño cambia dinámicamente y se asignan desde la reserva de memoria principal del núcleo. Cada página se divide en varios buffers de igual tamaño.
- **Los buffer_heads:** Son un conjunto de descriptores de buffers de caché. Contienen la información que el núcleo tiene

acerca de los buffers (el dispositivo al que pertenece el buffer, la distancia entre los datos dentro de ese dispositivo, y el tamaño del buffer).

Los buffers son guardados en diferentes tipos de listas: para buffers limpios, sucios, bloqueados y libres. Hay dos demonios en segundo plano que ayudan a escribir los buffers sucios. Uno se despierta a intervalos regulares, y escribe los buffers que han estado sucios por un determinado tiempo. El otro es un hilo del núcleo que se despierta cuando hay una proporción demasiado alta del caché de buffers que está sucia.

22.8.1.2 El gestor de solicitudes

Es la capa de software que controla la lectura y escritura del contenido de los buffers de y a un driver de dispositivo por bloque.
Se mantiene una lista individual de solicitudes para cada controlador de dispositivo por bloques, y las solicitudes se planifican según el algoritmo C-SCAN.
A medida que se hacen nuevas solicitudes de E/S, el gestor intenta fusionar las solicitudes de las listas por dispositivo, pasando de varias solicitudes pequeñas, a una grande.
Se pueden hacer solicitudes de E/S que pasen por alto el caché de buffers, y se usa generalmente en el núcleo, y en memoria virtual.

22.8.2 Dispositivos por caracteres

El núcleo prácticamente no preprocesa las solicitudes de leer o escribir en un dispositivo por caracteres, y se limita a pasar la solicitud al dispositivo en cuestión para que él se ocupe de ella.

22.9 Comunicación entre Procesos

22.9.1 Sincronización y señales

Para informar a un proceso que sucedió un suceso se utiliza una *señal*. Este sistema tiene la desventaja de que el número de señales es limitado y estas no pueden enviar información, solo puede comunicar a un proceso el hecho de que ocurrió un suceso.
El núcleo de Linux no usa señales para comunicarse con los procesos que se están ejecutando en modo de núcleo.
Generalmente hay una cola de espera para cada suceso, y cuando ese suceso se produce, despierta todos los procesos que están en esa cola.
Otra forma que utiliza Linux como comunicación son los semáforos.

22.9.2 Transferencia de datos entre procesos

Linux ofrece el mecanismo *pipe* (conducto) que permite a un proceso hijo heredar un canal de comunicación con su padre.
La memoria compartida ofrece un mecanismo extremadamente rápido para comunicar cantidades grandes o pequeñas de datos. Pero tiene como desventaja que no ofrece mecanismo de sincronización, por lo cual es recomendable usarla con algún otro método de sincronización.

22.10 Estructura de redes

Linux apoya tanto los protocolos estándar de Internet de UNIX y de otros SO. El trabajo de redes se implementa con tres capas de software:

- **Interfaz de sockets:** Las aplicaciones de usuario efectúan todas las solicitudes de trabajo con redes a través de esta interfaz.
- **Controladores de protocolos:** Cada vez que llegan datos de trabajo con redes a esta capa, se espera que los datos estén etiquetados con un identificador que indique el protocolo de red. Cuando termina de procesar los paquetes, los envía a la siguiente capa. Decide a que dispositivo debe enviar el paquete.

- **Drivers de dispositivos de red:** Codifican el tipo de protocolo de diferentes maneras dependiendo de su medio de comunicación.

Los paquetes de IP se entregan al controlador de IP. Este debe decidir hacia donde está dirigido el paquete, y remitirlo al driver de protocolo indicado. El software de IP pasa por *paredes cortafuegos*, filtrando los paquetes para fines de seguridad.
Otras funciones son el desensamblado y reensamblado de paquetes grandes, para poder encolarlos en dispositivos en *fragmentos* más pequeños si no pueden ser encolados por su tamaño.

22.11 Seguridad

Los problemas de seguridad se pueden clasificar en dos grupos:

- **Validación:** Asegurarse de que solo personas autorizadas puedan ingresar al sistema.
- **Control de acceso:** Proporcionar un mecanismo para verificar si un usuario tiene derechos suficientes para acceder a un objeto dado.

22.11.1 Validación

La contraseña de un usuario se combina con un valor aleatorio y resultado se codifica con una función de transformación unidireccional (esto implica que no es posible deducir la contraseña original a partir del archivo de contraseña) y se almacena en el archivo de contraseñas. Cuando el usuario la ingresa, se vuelve a codificar, y se busca en el archivo codificado para ver si coincide.

22.11.2 Control de acceso

Se realiza con la ayuda de identificadores numéricos únicos. El *uid* identifica un solo usuario o un solo conjunto de derechos de acceso, y un *gid* es un identificador de grupo. Cada objeto disponible en el sistema que esta protegido por los mecanismos de control de acceso por usuario y grupo tiene asociado un solo uid y un solo gid.
Linux efectúa el control de acceso a los objetos mediante una *mascara de protección*, que especifica cuales modos de acceso (lectura, escritura, o ejecución) habrán de otorgarse a procesos con acceso de propietario, acceso a grupos o al mundo. Otros de los mecanismos que ofrece es el setuid, el cual permite a un programa ejecutarse con privilegios distintos de los que tiene el usuario que ejecuta el programa.

23 Windows NT

Es un sistema operativo multitareas, expropiativo de 32 bits para microprocesadores modernos. NT se puede trasladar a varias arquitecturas, y hasta la versión 4 no es multiusuario.
Las dos versiones de NT son Windows NT Workstation y Windows NT Server. Ambas usan el mismo código para el núcleo y el sistema, pero el Server está configurado para aplicaciones cliente-servidor.

23.1 Historia

A mediados de los 80’s Microsoft e IBM cooperaron para desarrollar OS/2, escrito en lenguaje ensamblador para sistemas monoprocesador Intel 80286.
En 1988 Microsoft decidió desarrollar un sistema portátil que apoyara las interfaces de programación de aplicaciones (API). Se suponía que NT utilizaría la API de OS/2, pero utilizó la Win32 API.
Las primeras versiones fueron Windows NT 3.1 y 3.1 Advanced Server. En la versión 4, NT adoptó la interfaz con el usuario de Windows 95, e incorporó software de servidor de Web en Internet y de navegador. Se pasaron al núcleo algunas rutinas de interfaz con el usuario y código de gráficos a fin de mejorar el desempeño, con el efecto secundario de disminución en la confiabilidad del sistema.

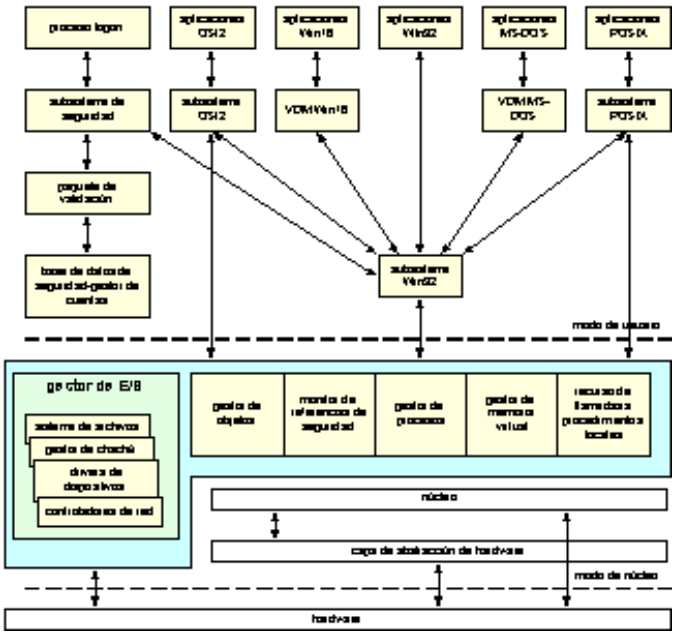
23.2 Principios de diseño

Los objetivos clave del sistema son transportabilidad, seguridad, soporte a multiprocesadores, extensibilidad, soporte internacional y compatibilidad con aplicaciones MS-DOS y MS-Windows.

- **Extensibilidad:** Se implementó con una arquitectura de capas a fin de mantenerse al día con los avances tecnológicos. El ejecutivo NT que opera en modo protegido, proporciona los servicios básicos, y encima operan varios subsistemas servidores en modo usuario, entre los cuales se encuentran los *subsistemas de entorno o ambiente* que emulan diferentes sistemas operativos. Es posible añadir nuevos subsistemas de entorno fácilmente, además NT utiliza controladores cargables, para poder cargar nuevos sistemas de archivos, dispositivos de E/S, etc, mientras el sistema está trabajando.
- **Transportabilidad:** Casi todo el sistema fue escrito en C y C++, al igual que UNIX, y todo el código que depende del procesador está aislado en una DLL llamada *capa de abstracción de hardware (HAL)*. Las capas superiores de NT dependen de la HAL y no del hardware, ayudando a la portabilidad.
- **Confiabilidad:** Emplea protección por hardware para la memoria virtual y por software para los recursos del SO. Posee también un sistema de archivos propio (NTFS) que se recupera automáticamente de muchas clases de errores del sistema de archivos después de una caída del sistema.
- **Compatibilidad:** Ofrece compatibilidad en el nivel de fuente (se compilan sin modificar el código fuente) con aplicaciones que cumplen el estándar IEEE 1003.1 (POSIX). Ejecuta también binarios de muchos programas compilados para Intel X86. Esto se logra mediante los subsistemas de entorno antes mencionados, aunque la compatibilidad no es perfecta, ya que por ejemplo, MS-DOS permitía accesos directos a los puertos de hardware y NT prohíbe tal acceso.
- **Desempeño:** Los subsistemas se comunican de forma eficiente llamando a procedimientos locales que transfieren mensajes con gran rapidez.
- **Uso internacional:** Cuenta con soporte para diferentes ubicaciones a través de la API de soporte a idiomas nacionales (NLS). El código de caracteres nativo es UNICODE, aunque reconoce caracteres ANSI, convirtiéndolos en UNICODE (8 bites a 16 bites).

23.3 Componentes del sistema

La arquitectura NT es un sistema de módulos en capas.
Una de las principales ventajas de este tipo de arquitectura es que las interacciones entre los módulos son relativamente sencillas.



23.3.1 Capa de abstracción de hardware

Es una capa de software que oculta las diferencias de hardware para que no las perciban los niveles superiores, y lograr portabilidad. Es usada por el núcleo y los drivers de dispositivos, por lo tanto solo se necesita una versión de cada driver, para las diferentes plataformas de hardware.
Por razones de desempeño, los controladores de E/S y de gráficos acceden directamente al hardware.

23.3.2 Núcleo

Constituye los cimientos del ejecutivo y los subsistemas. Siempre está cargado en memoria principal.
Tiene cuatro obligaciones principales:

- Planificación de hilos
- Manejo de interrupciones y excepciones
- Sincronización de bajo nivel del procesador

- Recuperación después de una caída en el suministro de electricidad

Está orientado a objetos. Los objetos están definidos por el sistema, y tiene atributos y métodos.

Windows NT, también maneja los conceptos de hilos y procesos. El proceso tiene un espacio de direcciones en memoria virtual, información sobre prioridad, y afinidad por uno o más procesadores.
Cada proceso tiene uno o más hilos, y cada hilo tiene su propio estado, que incluye prioridad, afinidad de procesador e información de contabilidad.

Estados de los hilos:

- **Listo:** Espera para ejecutarse.
- **A punto:** El hilo listo con la prioridad más alta se pasa a este estado, implicando que será el siguiente hilo en ejecutarse. Se mantiene un hilo a punto para cada procesador, en un sistema multiprocesador.
- **Ejecutándose:** Se ejecuta n un procesador hasta ser desalojado por un hilo de más alta prioridad, hasta terminar, hasta que se agote su cuanto de tiempo o hasta que necesite E/S.
- **Esperando:** Espera una señal, como por ejemplo el fin de E/S.
- **Transición:** Espera los recursos necesarios para ejecutarse.
- **Terminado:** Terminó su ejecución.

El despachador tiene un esquema de 32 niveles de prioridad, dividido en dos clases:

- **Tiempo real:** Hilos con prioridad de 16 a 31.
- **Clase variable:** Hilos con prioridad de 0 a 15.

El despachador usa una cola para cada prioridad, y las recorre de mayor a menor, buscando el primer hilo listo a ser ejecutado. Cuando se agota el cuanto de un hilo, se interrumpe, y si el hilo es de prioridad variable, la prioridad se reduce, aunque nunca por debajo de la prioridad base.
Cuando un hilo de prioridad variable sale del estado de espera, se aumenta su prioridad. Esta estrategia tiende a mejorar los tiempos de respuesta de los hilos interactivos que están usando el ratón y las ventanas, manteniendo los dispositivos de E/S ocupados por los hilos limitados por E/S, y que los hilos limitados por CPU, actúen en segundo plano con los ciclos de CPU sobrantes. Además, la ventana actual con la que el usuario está interactuando, recibe un aumento de prioridad para reducir su tiempo de respuesta.
Si un hilo de tiempo real, con prioridad más alta queda listo mientras se ejecuta un hilo con mas baja prioridad, éste es desalojado, por lo tanto la expropiación proporciona acceso preferencial al CPU a los hilos de tiempo real, sin embargo, no se pueden garantizar que se cumplan los límites de tiempo, y por ello NT no es un sistema de tiempo real duro.
El núcleo también maneja trampas para excepciones e interrupciones por hardware o software. NT define excepciones independientes de la arquitectura como violación de acceso a memoria, desbordamiento de enteros, desbordamiento hacia infinito o cero de punto flotante, división entera entre cero, división punto flotante entre cero, instrucción no válida, falta de alineación de datos, instrucción privilegiada, error de lectura de página, violación de página guardia, etc.
El manejador de trampas se encarga de excepciones sencillas, y el *despachador de excepciones* del núcleo se encarga de las demás, buscando manejadores que se encarguen de ellas. Cuando ocurre una excepción en modo de núcleo, si no se encuentra el manejador correcto, ocurre un error fatal del sistema, presentando la “pantalla azul” que significa, fallo del sistema.
El núcleo utiliza una *tabla de despacho de interrupciones* para vincular cada nivel de interrupción con una rutina de servicio, y en un computador multiprocesador, NT mantiene una tabla para cada procesador.
NT se puede ejecutar en máquinas con multiprocesadores simétricos, por lo tanto el núcleo debe evitar que dos hilos modifiquen estructuras de datos compartidas al mismo tiempo. Para lograr la mutua exclusión, el núcleo emplea cerraduras giratorias que residen en memoria global. Debido a que todas las actividades del procesador se suspenden cuando un hilo intenta adquirir una cerradura giratoria, se trata que sea liberada lo más rápido posible.

23.3.3 Ejecutivo

Presta un conjunto de servicios que los subsistemas de entorno pueden usar.

23.3.3.1 Gestor de objetos

NT es un sistema orientado a objetos, y el gestor supervisa su uso. Cuando un hilo desea usar un objeto, debe invocar el método **open** del gestor de objetos, para obtener un *handle* para ese objeto. Además el gestor se encarga de verificar la seguridad, verificando si el hilo tiene derecho de acceder a ese objeto. El gestor también se mantiene al tanto de que procesos están usando cada objeto. Se encarga de eliminar objetos temporales cuando ya no hay referencias a ellos (Nota: algo del estilo garbageCollector).

23.3.3.2 Nombres de objetos

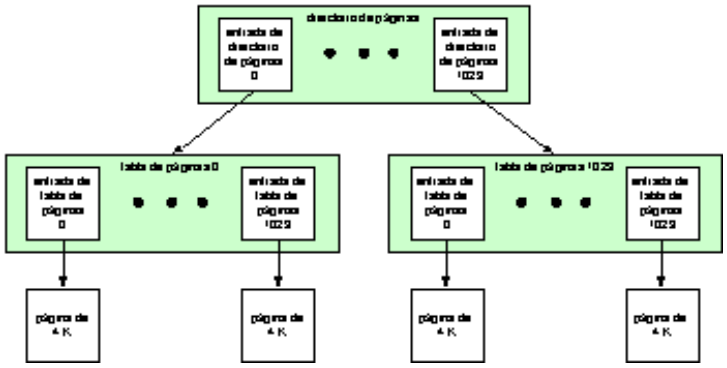
NT permite dar nombres a los objetos. El espacio de nombres es global, por lo tanto un proceso puede crear un objeto con nombre, y otro, puede hacer un handle hacia él, y compartirlo con el primer proceso. El nombre puede ser temporal o permanente. Uno permanente representa entidades, como unidades de disco que existe aunque ningún proceso lo esté usando. Los temporales existen mientras alguien haga referencia a ellos (Nota: no sobreviven al garbageCollector).
Los nombres de objeto tienen la misma estructura que los nombres de camino de archivos en MS-DOS y UNIX. Los directorios se representan con un *objeto directorio*, que contiene los nombres de todos los objetos de ese directorio.
El espacio de nombres de objetos aumenta al agregar un *dominio de objetos*, como por ejemplo discos flexibles y duros, ya que estos tienen su propio espacio de nombres que se debe injertar en el espacio de nombres existente.
Al igual que UNIX, NT implementa un *objeto de enlace simbólico*, refiriéndose a un mismo archivo con varios seudónimos o alias, como por ejemplo la correspondencia de nombres de unidades en NT, con las letras de unidad estándar de MS-DOS.
Cada proceso crea handle a varios objetos, los cuales son almacenados en una *tabla de objetos* del proceso, y al terminar, NT cierra automáticamente todos esos handle.

Cada objeto está protegido por una *lista de control de acceso* que contiene los derechos de acceso. El sistema los compara con los derechos de acceso del usuario que intenta acceder al objeto, y verifica si puede hacerlo. Esta verificación solo se hace al abrir el objeto, y no al crear un handle, por lo tanto los servicios internos de NT que usan handle, pasan por alto la verificación de acceso.

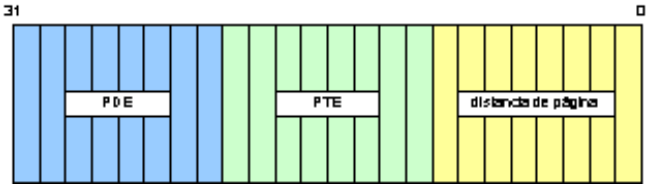
23.3.3.3 Gestor de memoria virtual

El diseñador del gestor, supone que el hardware apoya la correspondencia virtual a física, un mecanismo de paginación y coherencia de caché transparente en sistemas multiprocesador, y que permite hacer que varias entradas de la tabla de páginas, correspondan al mismo marco de página.
Utiliza un esquema de gestión basado en páginas con un tamaño de 4 KB, y usa direcciones de 32 bits, por lo tanto cada proceso tiene un espacio de direcciones virtual de 4GB. Los 2 GB superiores son idénticos para todos los procesos, y NT los usa en modo núcleo. Los 2GB inferiores son distintos para cada proceso y son accesibles en modo usuario o núcleo.
Dos procesos pueden compartir memoria obteniendo handle para el mismo objeto, pero sería ineficiente ya que los dos deberían reservar el espacio de memoria, antes de poder acceder al objeto. Para ello se creó el *objeto sección* que representa un bloque de memoria compartida, y que se maneja con handle.
Un proceso puede controlar el uso de un objeto en memoria compartida. Puede protegerse de modo lectura-escritura, solo lectura, solo ejecución, página guardia y copiar al escribir. La página guardia genera una excepción si es accedida, y sirve por ejemplo para controlar si un programa defectuoso va más allá del tope de un arreglo. El mecanismo de copiar al escribir, permite ahorrar memoria, ya que si dos procesos desean copias independientes de un mismo objeto, coloca una sola copia compartida en la memoria, activando la propiedad de copiar al escribir en esa región de memoria, y si uno de los objetos desea modificar los datos, el gestor hace primero una copia privada de la página para que ese proceso la use.

La traducción de direcciones virtuales de NT usa varias estructuras de datos. Cada proceso tiene un *directorio de páginas* que contiene 1024 *entradas de directorio de páginas (PDE)* de cuatro bytes cada una. El directorio es privado, pero podría compartirse entre procesos si fuera necesario.
Cada entrada de directorio, apunta a una *tabla de páginas* que contiene 1024 *entradas de tabla de páginas (PTE)* de cuatro bytes cada una, y cada *entrada de tabla de páginas* apunta a un *marco de página* de 4 KB en la memoria física.



Una dirección de memoria virtual de 32 bits se divide en tres enteros. Los primeros 10 son el índice de directorio de página, seleccionando la entrada al directorio de páginas que apunta a una tabla de páginas. Los siguientes 10 bits sirven para escoger una PTE de esa tabla de páginas, la cual apunta a un marco de página en la memoria física. Los restantes 12 bits se usan para describir la página.



La página puede estar en seis estados:

- **Válida:** Usada por un proceso activo.
- **Libre:** No se hace referencia en ninguna PTE.
- **En ceros:** Página libre, con ceros, lista para usarse de inmediato.
- **A punto:** Ha sido eliminada del conjunto de trabajo de un proceso.
- **Modificada:** Se ha escrito pero todavía no se copió en disco.
- **Defectuosa:** No puede utilizarse porque se detectó un error de hardware.

El principio de *localidad* dice que cuando se usa una página, es probable que en un futuro cercano, se haga referencia a páginas adyacentes, por lo tanto, cuando se produce un fallo de página, el gestor de memoria, también trae las páginas adyacentes, a fin de reducir nuevos fallos.
Si no hay marcos de página disponibles en la lista de marcos libres, NT usa una política de reemplazo FIFO por proceso para expropiar páginas.
NT asigna por defecto 30 páginas a cada proceso. Periódicamente va robando una página válida a cada proceso, y si el proceso se sigue ejecutando sin producir fallos por falta de página, se reduce el conjunto de trabajo del proceso, y se añade la página a la lista de páginas libres.

23.3.3.4 Gestor de procesos

Presta servicios para crear, eliminar y usar hilos y procesos. No tiene conocimiento de las relaciones padre-hijo ni de jerarquías de procesos.

23.3.3.5 Recurso de llamadas a procedimientos locales

El LPC (local procedure call) sirve para transferir solicitudes y resultados entre procesos clientes y servidores dentro de una sola máquina. En particular se usa para solicitar servicios de los diversos subsistemas de NT. Es un mecanismo de transferencia de mensajes. El proceso servidor publica un objeto *puerto de conexión* que es visible globalmente. Cuando un cliente desea servicios de un subsistema, abre un handle (mango) al objeto puerto de conexión de ese subsistema, y le envía una solicitud de conexión. El servidor crea un canal, y le devuelve un handle al cliente.
El canal consiste en un par de puertos de comunicación privados, uno para los mensajes del cliente al servidor, y otro para los del servidor al cliente.
Hay tres técnicas de transferencia de mensajes para el canal LPC:

- *Mensajes pequeños:* Para mensajes de hasta 256 bytes. Se usa la cola de mensajes del puerto como almacenamiento

intermedio, y los mensajes se copian de un proceso al otro.

- *Mensajes medianos:* Se crea un objeto *sección de memoria compartida* para el canal, y los mensajes del puerto, contienen un puntero, e información de tamaño referidos a ese objeto sección, evitando la necesidad de copiar mensajes grandes.
- *Mensajes LPC, o LPC rápida:* Es utilizado por las porciones de exhibición grafica del subsistema Win32. Cuando un cliente pide LPC rápida, el servidor prepara tres objetos: Un hilo servidor, dedicado a atender solicitudes, un objeto sección de 64KB y un objeto *par de sucesos*, que es un objeto de sincronización para Win32, que notifica cuando el hilo cliente copio un mensaje en el servidor o viceversa.
    - *Ventajas:*
        - El objeto sección elimina el copiado de mensajes, ya que representa memoria compartida.
        - El objeto par de sucesos elimina el gasto extra de utilizar el objeto para transferir punteros y longitudes.
        - El hilo servidor elimina el gasto extra de determinar que hilo cliente está llamando al servidor ya que hay un hilo servidor por cada cliente.
    - *Desventajas:*
        - Se utilizan más recursos que con los otros métodos, por lo tanto solo se usa para el gestor de ventanas e interfaces con dispositivos de gráficos.

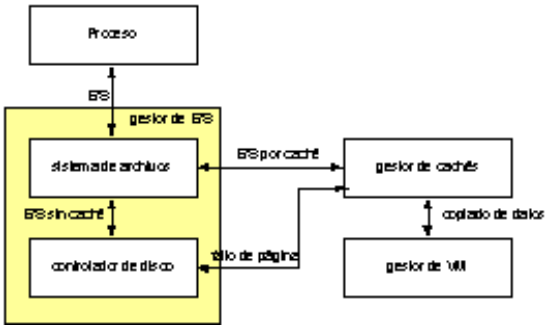
23.3.3.6 Gestor de E/S:

Funciones:

- Se mantiene al tanto de cuales sistemas de archivos instalables están cargados.
- Gestiona buffers para solicitudes de E/S
- Trabaja con el gestor de cachés de NT que maneja los cachés de todo el sistema de E/S.
- Trabaja con el gestor de Memoria Virtual para efectuar E/S de archivos con correspondencia en la memoria.
- Apoya operaciones de E/S, sincrónicas y asincrónicas.
- Proporciona tiempos límite para los controladores, y cuenta con mecanismos para que un controlador llame a otro.

En muchos sistemas operativos, el sistema de archivos se encarga de manejar los caché, en cambio NT cuenta con un manejo de cachés centralizado. El gestor de caché presta servicios, controlado por el gestor de E/S, e íntimamente vinculado con el gestor de VM.
Los 2GB superiores del espacio de direcciones de un proceso, son del sistema, y el gestor de VM asigna la mitad del espacio disponible al caché del sistema.
Cuando el gestor de E/S recibe una solicitud de lectura, envía un IRP al gestor de cachés (a menos que se solicite lectura sin uso de caché). El gestor de caché calcula la dirección de la solicitud, y pueden suceder dos cosas con el contenido de la dirección:

- **Es nula:** asigna un bloque al caché e intenta copiar los datos invocados.
    - Se puede realizar el copiado: La operación finaliza.
    - No se puede realizar el copiado: Se produjo un fallo de página, lo que hace que el gestor de VM envíe una solicitud de lectura sin uso del caché. El driver apropiado, lee los datos, y los devuelve al gestor de VM, que carga los datos en el caché, y los devuelve al invocador, quedando satisfecha la solicitud.
- **No es nula:** Se leen los datos y la operación finaliza.



Para mejorar el desempeño, el gestor mantiene un historial de solicitudes de lectura, e intenta predecir solicitudes futuras. Si detecta un patrón en las tres solicitudes anteriores, como acceso secuencial hacia delante o atrás, obtiene datos y los coloca en el caché, antes de que la aplicación haga la siguiente solicitud, evitando la espera de leer a disco.
El gestor también se encarga de indicar al gestor de VM que escriba de vuelta en disco el contenido del caché. Efectúa escritura retardada, ya que acumula escrituras durante 4 o 5 segundos, y luego despierta el hilo escritor. Si sucede que un proceso escribe rápidamente llenando las páginas libres del caché, el gestor está controlando, y cuando se llena gran parte de las páginas, bloquea los procesos que escriben datos, y despierta el hilo escritor.

En el caso de redes, para que no queden procesos bloqueados en la red (causando retransmisión por exceso de tiempo), se puede ordenar al gestor de caché que no deje que se acumulen muchas escrituras en el caché. Hay una interfaz DMA directa entre el caché y la red, evitando el copiado de datos a través de un buffer intermedio.

23.3.3.7 Gestor de referencias de seguridad

La naturaleza orientada a objetos de NT permite la validación de accesos durante la ejecución y verificaciones de auditoria para cada entidad del sistema.

23.4 Subsistemas de entorno

Son procesos en modo usuario, apilados sobre los servicios nativos del ejecutivo de NT, que permiten ejecutar programas escritos para otros sistemas operativos, incluidos Windows de 16 bits, MS-DOS, POSIX y aplicaciones basadas en caracteres para el OS/2 de 16 bits. Cada subsistema de entorno, proporciona una API.

NT usa el subsistema Win32 como entorno operativo principal. Si el ejecutable no es Win32 nativo, se verifica si ya se está ejecutando el subsistema de entorno apropiado, y si no se está ejecutando, se pone en marcha como proceso en modo usuario y Win32 le transfiere el control a dicho subsistema.
El subsistema de entorno no accede directamente al núcleo, contribuyendo a la robustez de NT, ya que se permite controlar la correctitud de los parámetros antes de invocar la rutina del núcleo. NT prohíbe a las aplicaciones, mezclar rutinas de API de diferentes entornos.
La caída de un subsistema, no afecta a los demás, salvo el caso de Win32 que es el gestor de todos los demás.
Win32 clasifica las aplicaciones como gráficas o de caracteres (se supone que las salidas se dirigen a una pantalla ASCII de 80x24). Cuando es de caracteres, transforma la salida, en una representación gráfica dentro de una ventana (Caso de la ventanita de MS-DOS, en el escritorio de Windows). Al ser una emulación, se puede transferir texto de pantalla entre ventanas, a través del portapapeles.

23.4.1 Entorno MS-DOS

MS-DOS no tiene la complejidad de los otros subsistemas de entorno NT. Una aplicación Win32 llamada *máquina dos virtual (VDM)* se encarga de proporcionar el entorno. La VDM tiene una *unidad de ejecución de instrucciones* para ejecutar o emular instrucciones Intel 486, también el ROM BIOS de MS-DOS y los servicios de interrupción de software “int 21”, así como drivers de dispositivos virtuales para pantalla, teclado y puertos de comunicación.
Está basada en el código fuente de MS-DOS 5.0 y proporciona a la aplicación al menos 620 KB de memoria. El shell es una ventana muy similar a un entorno MS-DOS.

Si NT se ejecuta en un procesador X86, las aplicaciones gráficas de MS-DOS se ejecutan en modo de pantalla completa, y las de caracteres en cualquiera de las dos opciones (completa o ventana). Si la arquitectura de procesador es otra, todas las aplicaciones se ejecutan en ventanas. En general, las aplicaciones MS-DOS que acceden directamente a hardware, no funcionan bajo NT.
Hay aplicaciones para MS-DOS que acaparan la CPU (ya que no era multitareas) como espera activa o pausas. El despachador de NT detecta los retardos, y los frena (haciendo que la aplicación no funcione correctamente).

23.4.1 Entorno Windows de 16 bits

Corre por cuenta de un VDM que incorpora software adicional llamado *windows on windows* que cuenta con las rutinas del núcleo de Windows 3.1, y las rutinas adaptadoras (stubs) para las funciones del gestor de ventanas. Las rutinas adaptadoras, invocan las subrutinas apropiadas de Win32. Las aplicaciones que dependen de la estructura interna del gestor de ventanas de 16 bits no funcionan, porque Windows on Windows no implementa realmente la API de 16 bits.
Windows on Windows puede ejecutarse en forma multitareas con otros procesos en NT, pero solo una aplicación Win16 puede ejecutarse a la vez. Todas las aplicaciones son de un solo hilo, y residen en el mismo espacio de direcciones, con la misma cola de entrada.

23.4.3 Entorno Win32

Ejecuta aplicaciones Win32 y gestiona toda la E/S de teclado, ratón y pantalla. Es robusto, y a diferencia de Win16, cada proceso Win32 tiene su propia cola de entrada.
El núcleo NT ofrece operación multitareas expropiativa, que permite al usuario terminar aplicaciones que han fallado o no se necesitan.
Win32 valida todos los objetos antes de usarlos, evitando caídas, el gestor de objeto, evita que se eliminen objetos que todavía se están usando, e impide su uso, una vez eliminados.

23.4.4 Subsistema POSIX

Está diseñado para ejecutar aplicaciones POSIX, basadas en el modelo UNIX. El estándar no especifica impresión, pero se pueden usar impresoras a través del mecanismo de redirección de NT.
Las aplicaciones POSIX tienen acceso a cualquier sistema de archivos NT, pero hace respetar los permisos tipo UNIX en los árboles de directorio. No apoya varios recursos de Win32 como trabajo en redes, gráficos, etc.

23.4.5 Subsistema OS/2

NT proporciona recursos limitados en el subsistema de entorno OS/2, aunque en sus el propósito general había sido proporcionar un entorno OS/2 robusto.
Las aplicaciones basadas en caracteres solo pueden ejecutarse bajo NT en computadores Intel x86. Las de modo real, se pueden ejecutar en todas las plataformas, usando el entorno MS-DOS.

23.4.6 Subsistemas de ingreso y seguridad

Antes de que un usuario pueda acceder a objetos NT, el subsistema de ingreso debe validarlo (login). Para ello el usuario debe tener una cuenta, y proporcionar la contraseña.
Cada vez que el usuario intenta acceder a un objeto, el subsistema chequea que se tengan los permisos correspondientes.

23.5 Sistema de archivos

Los sistemas MS-DOS utilizaban el sistema de archivos FAT, de 16 bits, que tiene muchas deficiencias, como fragmentación interna, una limitación de tamaño de 2 GB, y falta de protección de archivos. FAT32 resolvió los problemas de tamaño y fragmentación, pero su desempeño y funciones son deficientes en comparación con NTFS.
NTFS se diseñó teniendo en mente muchas funciones, incluidas recuperación de datos, seguridad, tolerancia de fallos, archivos y sistemas de archivos grandes, nombres UNICODE y compresión de archivos. Por compatibilidad también reconoce los sistemas FAT.

23.5.1 Organización interna

La entidad fundamental de NTFS es el volumen. El administrador de disco de NT crea los volúmenes que se basan en particiones de disco lógicas. Un volumen puede ocupar una porción de disco, un disco entero o varios discos.
No maneja sectores de disco individuales sino que usa *cúmulos* (número de sectores de disco que es potencia de 2) como unidad de asignación. El tamaño del cúmulo se configura cuando se formatea el sistema de archivos NTFS.

Tamaños por omisión:

- Tamaño de sector en el caso de volúmenes de hasta 512 Mb.
- 1 KB para volúmenes de hasta 1 GB
- 2 KB para volúmenes de hasta 2 GB
- 4 KB para volúmenes mayores

Este tamaño es mucho menor al del cúmulo de FAT, ayudando a reducir la fragmentación interna. Ej: En un disco de 1.6 GB con 16000 archivos, se podrían perder 400MB con sistema FAT (cúmulos de 32KB), o 17MB bajo NTFS.
NTFS usa *números de cúmulo lógicos (LCN),* y los asigna numerando los cúmulos desde principio de disco, al final, pudiendo calcular la distancia física en el disco, multiplicando el LCN por el tamaño del cúmulo.
Un archivo NTFS, no es un simple flujo de bytes como en MS-DOS o UNIX, sino un objeto estructurado con atributos, algunos estándar, y otros específicos para ciertas clases de archivos.
El espacio de nombres está organizado como una jerarquía de directorios, y cada directorio emplea una estructura llamada *arbol B+*, debido a que elimina el costo de reorganizar el árbol, y tiene la propiedad de que la longitud de cada camino desde la raíz hasta una hoja, es la misma. La *raíz índice* de un directorio, contiene el nivel superior del árbol. Si el directorio es grande, el nivel superior contiene punteros a alcances de disco que contienen el resto del árbol.
Cada entrada de directorio contiene el nombre y la referencia del archivo y una copia del tiempo de actualización, y del tamaño. Esto es útil para no tener que recabar en la MFT (*Tabla maestra de archivos*) cuando se pide un listado de los archivos del directorio.
Los datos del volumen NTFS se almacenan en archivos. El primero es la MFT, y el segundo contiene una copia de las 16 primeras entradas de la MFT, que se utiliza en la recuperación si la MFT se daña.

Otros archivos:

- **Archivo de bitácora:** Registra actualizaciones de los datos del sistema de archivos.
- **Archivo de volumen:** Contiene el nombre del volumen, la versión de NTFS que dio formato, y un bit que indica si es posible que el volumen se haya corrompido, y se necesite verificar su consistencia.
- **Tabla de definición de atributos:** Indica los atributos del volumen y que operaciones se pueden efectuar con cada uno de ellos.
- **Directorio raíz:** Es el directorio del nivel más alto en la jerarquía del sistema de archivos.
- **Archivo de mapa de bits:** Indica que cúmulos del volumen están asignados, y cuales están libres.
- **Archivo de arranque:** Contiene el código de inicio de NT, y debe estar en una dirección específica para que el autoarranque de la ROM lo localice fácilmente. También contiene la dirección de la MFT.
- **Archivo de cúmulos defectuosos:** Sigue la pista a las áreas defectuosas del volumen, y se usa para recuperación de errores.

23.5.2 Recuperación

En NTFS, todas las actualizaciones de estructura de datos, se efectúan en *transacciones.* Antes de alterar la estructura, se escribe un registro de bitácora con información para deshacerla y rehacerla, y luego de modificarla con éxito, se escribe un registro de confirmación en la bitácora.
Después de una caída el sistema puede restaurar las estructuras, procesando la bitácora, rehaciendo las operaciones de las transacciones confirmadas, y deshaciendo la de las que no lograron confirmarse antes de la caída.
Periódicamente (cada cinco segundos aproximadamente), se escribe en la bitácora un checkpoint, indicando al sistema que no necesita registros de bitácora anteriores a él, desechando los registros, para que la bitácora no crezca indefinidamente.
La primera vez que se accede a NTFS luego de una caída, se realiza automáticamente la recuperación del sistema de archivos. Esto no garantiza la correctitud de los archivos de usuario, solo asegura las estructuras de datos del sistema de archivos.
Si el espacio de la bitácora se reduce demasiado, NTFS suspende todas las E/S nuevas, para poder confirmar las que se están haciendo, poner un checkpoint en la bitácora, y borrar todos los datos ya confirmados.

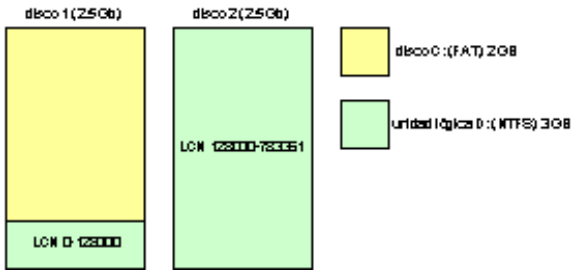
23.5.3 Seguridad

La seguridad se deriva del modelo de objetos, ya que cada uno tiene almacenado un descriptor de seguridad.

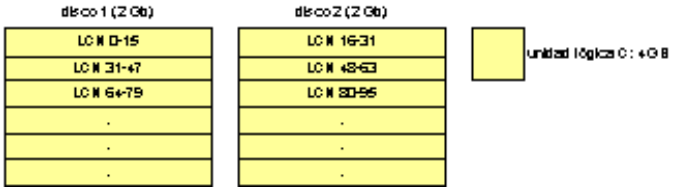
23.5.4 Gestión de volúmenes y tolerancia de fallos

FtDisk es el controlador de disco tolerante a fallos. Ofrece mecanismos para combinar múltiples unidades de disco en un solo

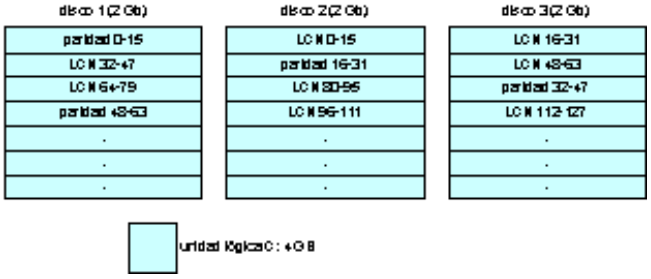
volumen lógico. Hay varias formas de hacerlo.
Se pueden concatenar discos lógicamente para formar un volumen lógico grande. En NT es llamado *conjunto de volúmenes* y puede consistir de hasta 32 unidades físicas. NTFS usa el mismo mecanismo LCN que para un solo disco físico, y FtDisk se encarga de hacer la correspondencia.



Otra forma de combinar múltiples particiones físicas es intercalar sus bloques bajo un régimen de turno



circular, para formar un *conjunto de franjas*. FtDisk usa 64Kb como tamaño de franja. Los primeros 64Kb de volumen lógico se almacenan en la primer partición física, los segundos en la segunda y así hasta que cada partición física tiene 64Kb de la lógica. Esto permite mejorar el ancho de banda de E/S ya que las unidades pueden transferir datos en paralelo.
Se pueden dividir también en *franjas de paridad*. Si hay 8 discos, entonces por cada siete franjas de datos en siete discos diferentes hay una franja de paridad en el octavo que contiene el **or exclusivo** de las siete franjas, por lo tanto, si una se destruye, se puede reconstruir. De todas formas, cada actualización de una franja de datos, obliga a recalcular la franja de paridad, así siete escrituras concurrentes, requieren calcular siete franjas de paridad, por lo tanto, se deben encontrar en discos diferentes, para no sobrecargar un mismo disco con el cálculo de todas las paridades. Las paridades se asignan entre los discos por turno circular.



Un esquema más robusto, es el *espejo de discos*. Un *juego de espejos* comprende dos particiones del mismo tamaño, en dos discos, con su contenido de datos idéntico. Cuando se requiere escritura, FtDisk escribe en los dos discos, y si hay fallo, se tiene una copia almacenada a salvo. Los dos discos, deben estar conectados a dos controladores de disco diferentes, por si el fallo es en el controlador. Esta organización se denomina *juego duplex*.

Para superar la situación en que sectores del disco se arruinan, FtDisk usa una técnica de hardware llamada *ahorro de sectores*, y NTFS, una de software llamada *retransformación de cúmulos.*
Cuando se da formato al disco, se crea correspondencia entre números de bloque lógicos y sectores libres de defectos del disco, dejando algunos sectores sin correspondencia como reserva. Si un sector falla, FtDisk ordena a la unidad de disco que lo sustituya por una unidad de reserva.
La retransformación de cúmulos se aplica por NTFS, que reemplaza un bloque defectuoso por uno no asignado, modificando los punteros pertinentes en la MFT, además tiene en cuenta que ese bloque no vuelva a ser asignado.

Si un bloque se arruina, se pierden datos, que en casos se pueden calcular si se tiene un disco espejo, o calculando la paridad por or exclusivo, y los datos reconstruidos se colocan en un nuevo espacio, que se obtuvo por ahorro de sectores o retransformación de cúmulos.

23.5.5 Compresión

NTFS puede aplicar compresión a archivos. Divide sus datos en *unidades de compresión*, que son bloques de 16 cúmulos contiguos. Cada vez que se escribe en una unidad de compresión, se aplica un algoritmo de compresión de datos. Para mejorar la lectura, NTFS las preobtiene y descomprime adelantándose a las solicitudes de las aplicaciones.
En caso de archivos *poco poblados* (contiene gran cantidad de ceros), la técnica es no almacenar en disco los cúmulos que contienen únicamente ceros. Se dejan huecos en la secuencia de números almacenados en la tabla MFT para ese archivo, y al leerlos, si NTFS encuentra un hueco en los números, llena con ceros esa porción del buffer del invocador. UNIX emplea la misma técnica.

23.6 Trabajo con redes

NT cuenta con recursos de gestión de redes, y apoya el trabajo de cliente-servidor. Además también tiene componentes de trabajo con redes como:

- Transporte de datos
- Comunicación entre procesos
- Compartimiento de archivos a través de una red
- Capacidad para enviar trabajos de impresión a impresoras remotas.

23.6.1 Protocolos

Implementa los protocolos de transporte como controladores, los cuales puede cargar y descargar dinámicamente, aunque en la práctica, casi siempre debe reiniciarse el sistema después de un cambio.

23.6.2 Mecanismos de procesamiento distribuido

NT no es distribuido, pero si apoya aplicaciones distribuidas. Mecanismos apoyados:

- **Aplicaciones NetBIOS:** Se comunican en NT a través de la red utilizando algún protocolo.
- **Conductos con nombre:** Son mecanismos de envío de mensajes, orientados a conexiones. Un proceso también puede usarlos para conectarse con otro proceso en la misma máquina. Puesto que el acceso a estos conductos es a través de la interfaz con el sistema de archivos, se aplican los mecanismos de seguridad sobre ellos.
- **Ranuras de correo:** Son un mecanismo de envío de mensajes sin conexiones. No son confiables, porque el mensaje podría perderse antes de llegar al destinatario.
- **Sockets:** Winsock es la API de sockets para ventanas. Es una interfaz de capa de sesión, compatible (en buena parte) con sockets de UNIX.
- **Llamada a procedimiento remoto (RPC***):* Es un mecanismo cliente-servidor que permite a una aplicación de una máquina, emitir una llamada a un procedimiento cuyo código está en otra máquina. El cliente invoca, un stub empaca sus argumentos en un mensaje y lo envía por la red a un proceso servidor específico. El servidor desempaca el mensaje, invoca el procedimiento, empaca los resultados en otro mensaje, y lo envía de vuelva al cliente, que espera el mensaje bloqueado. El cliente se desbloquea, lo recibe, y desempaca los resultados.
- **Lenguaje de definición de interfaces de Microsoft:** Se utiliza para describir los nombres de procedimientos remotos, argumentos y resultados, para facilitar las llamadas a procedimientos remotos. El compilador para este lenguaje genera archivos de cabecera que declaran rutinas adaptadoras para los procedimientos remotos, asi como un desempacador y un despachador.
- **Intercambio dinámico de datos (DDE):** Mecanismo para la comunicación entre procesos creado para Microsoft Windows.

23.6.3 Redirectores y servidores

En NT, una aplicación puede usar la API de E/S para acceder a archivos remotos. Un *redirector* es un objeto del lado del cliente que reenvía solicitudes de E/S a archivos remotos, donde un *servidor* las atiende. Por razones de desempeño y seguridad, los redirectores y servidores se ejecutan en modo de núcleo. Por razones de transportabilidad, redirectores y servidores utilizan la API TDI para el transporte por redes.

23.6.4 Dominios

Para gestionar derechos de acceso globales (ej: todos los usuarios de un grupo de estudiantes), NT utiliza el concepto de dominio.
Un dominio NT, es un grupo de estaciones de trabajo y servidores NT que comparten una política de seguridad y una base de datos de usuarios comunes. Un servidor de dominio controla y mantiene las bases de datos de seguridad. Otros servidores pueden actuar como controladores de respaldo, y validación de solicitudes de inicio.

A menudo es necesario manejar múltiples dominios dentro de una sola organización. NT ofrece cuatro modelos de dominios para hacerlo.

- **Modelo de dominio único:** Cada dominio queda aislado de los demás.
- **Modelo de dominio maestro:** Todos los dominios confían en un dominio maestro designado, que mantiene las bases de datos de cuentas de usuario, y proporciona servicios de validación. Puede apoyar hasta 40000 usuarios.
- **Modelo de múltiples dominios maestros:** Diseñado para organizaciones grandes, con más de un dominio maestro, y todos los dominios confían unos en otros. El usuario con cuenta en un dominio, puede acceder a recursos de otro dominio, si ese dominio certifica los permisos del usuario.
- **Modelo de múltiple confianza:** No existe un dominio maestro, todos los dominios confían unos en otros.

23.6.5 Resolución de nombres en redes TCP/IP

Es el proceso de convertir un nombre de computador en una dirección de IP. Por ejemplo, transformar www.bell-labs.com en 135.104.1.14.
NT cuenta con varios métodos, incluidos el Servicio de Nombres de Internet de Windows (WINS), resolución de nombres difundida, sistema de nombres de dominio (DNS).

WINS: Dos o más servidores WINS, mantienen una base de datos dinámica de vinculaciones Nombre-dirección de IP, y software cliente para consultarlos. Se usan dos servidores para que el servicio funcione, aunque falle uno, y además para distribuir la carga.

23.7 Interfaz con el programador

La API Win32 es la interfaz fundamental con las capacidades de NT.

23.7.1 Acceso a objetos del núcleo

Se acceden para obtener los servicios del núcleo. Cuando un proceso quiere acceder a un objeto X del núcleo, debe hacer **createX** y abre un handle a ese objeto. Para cerrar el handle hace **closeHandle** y el sistema elimina el objeto si nadie tiene handles con él.

23.7.1.1 Compartimiento de objetos

Hay tres formas:

- Un proceso hijo hereda un handle del padre. El padre y el hijo podrán comunicarse a través del objeto.
- Un proceso crea un objeto con nombre, y el segundo proceso lo abre con ese nombre.
  - Desventajas:
    - NT no ofrece un mecanismo para verificar si existe un objeto con nombre escogido.
    - El espacio de nombres es global. Por ejemplo, dos aplicaciones podrían crear un objeto llamado pepe, cuando se desean dos objetos individuales (y diferentes).
  - Ventajas:
    - Procesos no relacionados entre si, pueden compartir objetos fácilmente.
- Compartir objetos por medio de la función **DuplicateHandle**. Pero requiere de un método de comunicación entre procesos para pasar el mango duplicado.

23.7.2 Gestión de procesos

Un proceso es un ejemplar de una aplicación que se está ejecutando y un hilo es luna unidad de código que el sistema operativo puede planificar. Un proceso tiene uno o más hilos.
Una aplicación multihilada necesita protegerse contra acceso no sincronizado, y la función envolvente **beginthreadex** (iniciar

ejecución de hilo) proporciona la sincronización apropiada.

Las prioridades en el entorno Win32 se basan en el modelo de planificación NT, pero no pueden escogerse todos los valores de prioridad. Se usan cuatro clases:

- **IDLE_PRIORITY_CLASS:** Nivel de prioridad 4 (ocioso).
- **NORMAL_PRIORITY_CLASS:** Nivel de prioridad 8 (normal).
- **HIGH_PRIORITY_CLASS:** Nivel de prioridad 13 (alta).
- **REALTIME_PRIORITY_CLASS:** Nivel de prioridad 24 (tiempo real).

Solo los usuarios con *privilegio de incremento de prioridad* pueden pasar un proceso a la prioridad de tiempo real. Por defecto los administradores o usuarios avanzados tienen ese privilegio.
Cuando un usuario está ejecutando un programa interactivo, el sistema debe darle mayor prioridad, para ello tiene una regla de planificación que distingue entre el *proceso de primer plano* que es el seleccionado actualmente en la pantalla y los *procesos de segundo plano* que no están seleccionados. Cuando un proceso pasa a primer plano, el sistema incrementa su cuanto de planificación al triple, por lo que el proceso se ejecuta por un tiempo tres veces mayor, antes de ser desalojado.
Para sincronizar el acceso concurrente de hilos a objetos compartidos, el núcleo proporciona objetos de sincronización como semáforos y mutexes. Otro método es la *sección crítica*, que es una región de código sincronizada que solo un hilo puede ejecutar a la vez.

Una *fibra* es código en modo usuario, planificado según un algoritmo definido por el usuario. Se utiliza de la misma forma que los hilos, pero a diferencia de ellos, no se pueden ejecutar dos fibras concurrentemente, aún en un hardware multiprocesador.

23.7.3 Comunicación entre procesos

Una forma es compartiendo objetos del núcleo, y otra es transfiriendo mensajes.
Los mensajes se pueden publicar o enviar. Las rutinas de publicación son asincrónicas, por lo tanto envían el mensaje y regresan de inmediato. El hilo que envió el mensaje no sabe si llegó. Las rutinas de envío son sincrónicas, y bloquean al invocador hasta que se ha entregado y procesado el mensaje.

Cada hilo Win32 tiene su propia cola de entrada, y esta estructura es más confiable que la cola de entradas compartida de las ventanas de 16 bits, porque una aplicación bloqueada no bloquea a las demás.
Si un proceso que envió un mensaje, no recibe respuesta en cinco segundos, el sistema marca la aplicación como “No responde”.

23.7.4 Gestión de memoria

Hay varios mecanismos de uso de memoria como:

- **Memoria Virtual:** Para pedir y liberar memoria virtual se usa **VirtualAlloc** y **VirtualFree**. La función **VirtualLock** asegura páginas en memoria física, pudiendo asegurar 30 como máximas.
- **Archivos con correspondencia en memoria:** Es cómodo para que dos procesos compartan memoria.
- **Montículo (heap):** Es una región de espacio de direcciones reservada. Cuando Win32 crea un proceso, lo hace con un *montículo por omisión* de 1MB. El acceso al montículo es sincronizado para proteger las estructuras de datos de hilos concurrentes. Se usan funciones como HeapCreate, HeapAlloc, HeapRealloc, HeapSize, HeapFree y HeapDestroy.

Índice

Índice