Conceptos de Sistemas Operativos

Este apunte pertenece al libro "Operating System Concepts" de Abraham Silberschatz (quinta edición) con retoques del libro "Sistemas Operativos: diseño e implementación" de Tanenbaum.

I. Vistazo

El software de la computadora se puede dividir en dos tipos: los *programas del sistema*, que controlan la operación de la computadora; y los *programas de aplicación*, que realizan las tareas reales que el usuario desea. El programa del sistema más fundamental es el **sistema operativo**, que controla todos los recursos de la computadora y establece las bases sobre las que puede escribirse un programa de aplicación.
Un sistema operativo es un programa que actúa como intermediario entre el usuario de una computadora y el hardware de la computadora. El propósito de un sistema operativo es el de proveer un ambiente en el cual un usuario pueda ejecutar programas de una manera conveniente y eficiente.
El sistema operativo debe asegurar la correcta operación de la computadora. Para prevenir que los programas del usuario se interfieran con las operaciones propias del sistema, el hardware debe proveer mecanismos apropiados para asegurar tal comportamiento correcto.

1. Introducción

Un sistema operativo es un programa que actúa como intermediario entre los usuarios y el hardware de la computadora. El propósito de un sistema operativo es el de proveer un ambiente en el cual los usuarios puedan ejecutar programas. En este primer capítulo veremos el desarrollo que tuvieron los sistemas operativos desde los primeros sistemas interpretados hasta los actuales sistemas multiprogramados y de tiempo compartido.

¿Qué es un Sistema Operativo?

El sistema de una computadora se puede dividir en 4 partes principales:

- el hardware
- el sistema operativo
- los programas de aplicación
- los usuarios

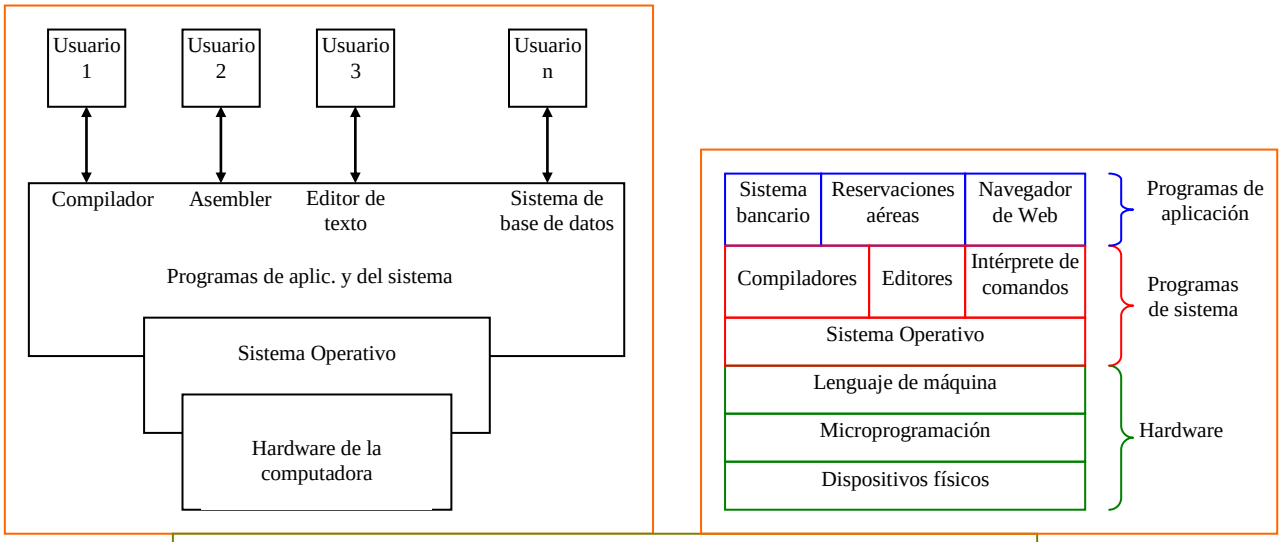


Figura 1.1 Vista abstracta de los componentes de una computadora.

El hardware (es decir, la unidad central de proceso –CPU-, la memoria, y los dispositivos de entrada/salida) provee los recursos básicos. La capa de microprograma del hardware existe sólo en algunas máquinas. Su función es obtener las instrucciones de lenguaje de máquina como ADD, MOVE y JUMP y las ejecuta en una serie de pasos pequeños. Por ejemplo, para ejecutar una instrucción ADD, el microprograma debe determinar donde se encuentran los números a sumar, obtenerlos, sumarlos y almacenar el resultado en algún lugar. Por otra parte, el conjunto de instrucciones que el microprograma interpreta define el lenguaje de máquina, que no es realmente parte de la máquina física pero normalmente se toma como tal.
Encima del sistema operativo está el resto del software de sistema. Aquí se encuentran el intérprete de comandos (shell), sistema de ventanas, compiladores, editores, y otros programas similares independientes de la aplicación. Si bien éstos programas son provistos por el fabricante de la computadora, no forman parte del sistema operativo. El sistema operativo es la porción de software que se ejecuta en modo kernel o monitor, mientras que los compiladores y los editores se ejecutan en modo usuario (un usuario está en libertad de escribir su propio compilador si es que lo desea, pero no puede escribir su propio manejador de interrupciones).
Los programas de aplicación son o comprados o escritos por los usuarios para resolver sus problemas particulares (tales como sistemas de base de datos, juegos, y programas de negocios).
Los programas de aplicación y los del sistema definen las formas en el cual los recursos son usados para resolver los problemas de los usuarios. Existen muchos tipos de usuarios (personas, maquinas, otras computadoras) tratando de resolver diferentes problemas. Cada tipo de usuario puede requerir que se resuelvan varias tareas. El sistema operativo controla y coordina el uso del hardware entre los diferentes programas de aplicación de los diferentes usuarios.
Los componentes de una computadora son el hardware, software y los datos. El sistema operativo sería un distribuidor de recursos. Una computadora tiene muchos recursos (entre el hardware y software) que pueden ser requeridos para resolver un problema: tiempo de CPU, espacio de memoria, espacio de los discos, dispositivos de entrada/salida, y otros. El sistema operativo actúa como un administrador de estos recursos y los distribuye entre los diferentes programas y usuarios para resolver sus tareas.
Un sistema operativo puede verse también como un programa de control. Un programa de control controla la ejecución de los diferentes programas de los usuarios para prevenir errores y el uso incorrecto de la computadora. Esto esta muy relacionado con la operación y control de los dispositivos de entrada/salida. Como el hardware por sí solo es muy difícil de utilizar existen programas que hacen fácil su uso. Estos programas son los programas de aplicación. Un ejemplo de estos programas puede ser aquellos que controlan los dispositivos de entrada/salida. Las funciones comunes de control y distribución de recursos están todas unidas en una sola pieza de software: el sistema operativo.
Una común definición de lo que es un sistema operativo es que es un programa corriendo todo el tiempo en la computadora (usualmente llamado kernel). Este programa trata de que el uso de la computadora para los usuarios sea más fácil.
En un principio los sistemas operativos fueron desarrollados para facilitar el uso del hardware. Veamos a continuación una breve vista de cómo se fueron desarrollando los sistemas operativos.

Sistema Batch simple

En un principio las computadoras eran físicamente máquinas enormes que corrían desde una consola. Los dispositivos de entrada comunes eran los lectores de tarjetas y drives de cinta. Los dispositivos de salida comunes eran impresoras, drives de cinta y perforadores de tarjetas. Los usuarios de tales sistemas no interactuaban directamente con el sistema de la computadora. En lugar de ello, el usuario preparaba un trabajo (que constaba del programa, los datos y alguna información de control) y se lo presentaba al operador de la computadora. El trabajo generalmente eran tarjetas perforadas. Un tiempo más tarde (minutos, horas o quizá días) se obtenía la salida. La salida constaba del resultado del programa o de memoria y registros vacíos en caso de que se haya producido un error. En esta era, los sistemas operativos eran bastantes simples ya que solamente se encargaba de pasar el control de un trabajo a otro. El sistema operativo siempre se encontraba en memoria.

Para aumentar la velocidad de proceso, los trabajos con necesidades similares se cargaban juntos y se ejecutaban como un grupo. Ante esto, el programador le dejaba el programa al operador, el operador ordenaba los programas en lotes (batches) con requerimientos similares, y cuando la computadora estaba disponible, ejecutaba cada lote. La salida de cada trabajo era enviada al programador correspondiente.
Así, un sistema de lotes normalmente leía un stream de trabajos separados (por ejemplo, de una lector de tarjetas), cada uno con su propia tarjeta de control que predefinia qué hacia el trabajo. Cuando se completaba el trabajo, su salida era usualmente impresa. En estos sistemas, la característica más importante era la falta de interacción entre el usuario y el trabajo mientras el trabajo se ejecutaba.
En este ambiente de ejecución, la CPU estaba normalmente ociosa. Esta ociosidad se producía porque la velocidad de los dispositivos mecánicos de entrada/salida era mucho más lenta que la velocidad de los dispositivos electrónicos. Con el tiempo se desarrollaron dispositivos de entrada/salida más rápidos pero, desafortunadamente, la velocidad de la CPU se ha incrementado aun más rápido, por lo que no solo el problema no fue resuelto sino que es más notorio.
La introducción a la tecnología de disco ha ayudado a este problema. En lugar de que la información leída de las tarjetas desde el lector de tarjetas vaya directamente a memoria, y luego el trabajo sea procesado, la información leída de las tarjetas desde el lector de tarjetas era llevada directamente a disco. La ubicación de cada tarjeta en disco era mantenida en una tabla administrada por el sistema operativo. Cuando un trabajo debía ser ejecutado, el sistema operativo satisfacía sus pedidos leyendo la parte de disco perteneciente a su tarjeta. Similarmente, cuando el trabajo hacia el pedido de la impresora para proyectar su salida, la salida se almacenaba en un sistema de buffer y luego era llevada a disco. Cuando el trabajo se completaba, la salida se imprimía. Esta forma de procesamiento se le llama spooling (figura 1.3). De hecho, el sistema spooling utiliza el disco como un enorme buffer, en la manera que leía el programa desde el disco y de la manera que almacenaba la salida del trabajo en disco hasta que la impresora este lista para aceptar el trabajo.
El sistema spooling también es utilizado para el procesamiento de datos de sitios remotos. La CPU envía los datos a una impresora remota (o acepta un trabajo desde un lector remoto). El procesamiento remoto se hace a la velocidad de la CPU remota, sin la intervención de la CPU local. La CPU local solo necesita ser notificada cuando se completa el procesamiento.
El spooling solapa la I/O de un trabajo con el cálculo de otros trabajos. Aún en un sistema simple, el spooler puede estar leyendo la entrada de un trabajo mientras imprime la salida de otro trabajo diferente. Durante el mismo tiempo, aún otro trabajo (o trabajos) puede estar siendo ejecutado.

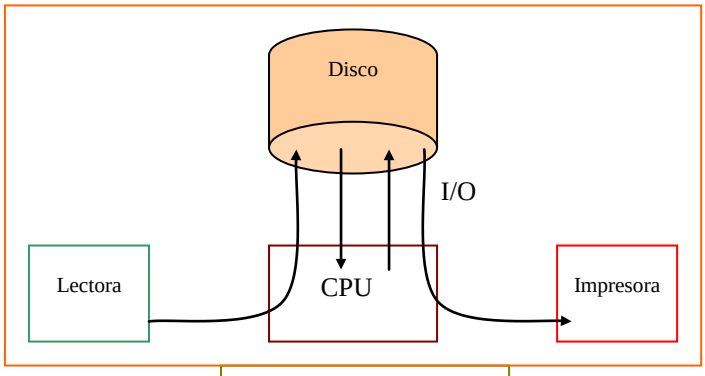


Figura 1.3 Spooling.

Sistema Batch multiprogramado

Spooling provee una importante estructura de datos: una pileta de trabajos. Spooling generalmente resultaba en varios trabajos que ya han sido leídos, esperando en discos, listos para correr. Una pileta de trabajos en disco permite al sistema operativo seleccionar cual será el próximo trabajo a correr, para incrementar la utilización de la CPU. Cuando los trabajos eran leídos directamente desde el lector de tarjetas o de cintas magnéticas, no era posible correr varios trabajos en diferente orden. Los trabajos

debían ser corridos secuencialmente, uno a la vez. Sin embargo, cuando varios trabajos están en dispositivos de acceso directo, tales como disco, puede ser posible el job scheduling.
El aspecto más importante del job scheduling es la habilidad para multiprogramar. Las operaciones del spooling tienen sus limitaciones. Un único usuario no puede, en general, mantener la CPU o el dispositivo de entrada/salida ocupado todo el tiempo. La multiprogramación incrementa la utilización de la CPU organizando los trabajos de manera que la CPU siempre tenga un trabajo para ejecutar.
La idea es la siguiente: el sistema operativo mantiene varios trabajos en memoria al mismo tiempo (figura 1.4). Este conjunto de trabajos es un subconjunto de todos los trabajos que están en la pileta de trabajos (ya que el número de trabajos que pueden estar almacenados en memoria es mucho menor que la cantidad de trabajos que pueden estar en la pileta de trabajos). El sistema operativo elige un trabajo de la memoria y comienza a ejecutarlo. Eventualmente, el trabajo puede necesitar esperar por alguna tarea, tal como la completitud de una operación de I/O. En un sistema que no es multiprogramado, la CPU quedaría ociosa. En un sistema multiprogramado, el sistema operativo cambia de trabajo y comienza la ejecución de otro. Cuando este último trabajo necesita esperar, la CPU es "cambiada" a otro trabajo, etc. Eventualmente, el primer trabajo puede finalizar su espera y conseguirá que se le asigne otra vez la CPU. Mientras haya trabajos para ejecutar, la CPU nunca quedara ociosa.

Figura 1.4 Memoria en un sistema multiprogramado.

En los sistemas operativos multiprogramados, todos los trabajos que entran al sistema son almacenados en una pileta de trabajos. Esta pileta consiste de todos los procesos residentes en los dispositivos de almacenamiento masivo esperando por su almacenamiento en memoria. Si varios trabajos son leídos para ser almacenados en memoria y en dicha memoria no hay lugar para todos ellos, entonces el sistema operativo debe elegir entre ellos. Esta decisión es realizada por el job scheduling que más adelante se verá con más detalle (capítulo 5). Cuando el sistema operativo selecciona un trabajo de la pileta de trabajos, éste lo carga en la memoria para su ejecución. El hecho de tener varios programas en memoria provoca tener un administrador de la memoria el cual se verá también más adelante (capítulo 8 y 9). Además, si existen varios trabajos listos para su ejecución en un tiempo dado, el sistema debe elegir uno de entre ellos. Esta elección es realizada por el CPU scheduling (capítulo 5). Por último, múltiples trabajos corriendo concurrentemente requieren una gran administración para que un trabajo no afecte la ejecución de otro. Todos estos temas se verán más adelante.

Sistemas de tiempo compartido

Los sistemas de lotes multiprogramados proveen un ambiente donde los diferentes recursos del sistema (por ejemplo, CPU, memoria, dispositivos periféricos) son utilizados efectivamente. Sin embargo, existen algunas dificultados con los sistemas por lotes desde el punto de vista del usuario. Ya que el usuario no pueden interactuar con los trabajos cuando éstos están siendo ejecutados, el usuario debe establecer la tarjeta de control para que maneje todas las posibles salidas. En un trabajo multi-etapa, los subsecuentes pasos pueden depender de los resultados que se obtuvieron en etapas anteriores. Por

ejemplo, la ejecución de un programa puede depender del éxito de la compilación. Puede ser complicado definir completamente que hacer en todos los casos.
Otra dificultad es que los programas deben ser depurados estáticamente. Un programador no puede modificar un programa en su ejecución para estudiar su comportamiento.
El tiempo compartido, o multitarea, es una extensión de la multiprogramación. Múltiple trabajos son ejecutados por la CPU alternando entre ellos, pero la alternación ocurre tan frecuentemente de manera que el usuario puede interactuar con cada programa mientras éste esta corriendo.
En un sistema interactivo, el sistema provee una comunicación on-line entre el usuario y el sistema. El usuario da instrucciones al sistema operativo, o directamente al programa, y recibe una respuesta inmediata. Usualmente, un teclado es utilizado para proveer la entrada, y un monitor es utilizado para proveer la salida. Cuando el sistema operativo finaliza la ejecución de un comando, este busca la siguiente sentencia de control, no desde el lector de tarjetas sino desde el teclado del usuario. El usuario da un comando, esperando por la respuesta, y decide el siguiente comando basado en el resultado del comando previo.
Si los usuarios son capaces de acceder tanto a los datos como al código, de debe disponer de un file-system on-line. Un archivo es una colección de información relacionada definida por su creador. Comúnmente, los archivos representan programas (tanto en su forma fuente como en su forma objeto) y datos. Los archivos de datos pueden ser numéricos, alfabéticos, o alfa-numéricos. Los archivos pueden ser de forma libre, tal como archivos de texto, o pueden tener un formato rígido. Los archivos son organizados en directorios, los cuales hacen fácil localizarlos y acceder a ellos.
Los sistemas batches son apropiados para la ejecución de grandes trabajos que necesitan poca interacción. El usuario puede suministrar el trabajo y retornar más tarde por la respuesta, por lo que no es necesario que el usuario se quede esperando la respuesta mientras el trabajo esta siendo ejecutado. Los trabajos interactivos tienden a estar compuestos por muchas pequeñas acciones, donde el resultado del siguiente comando puede ser impredecible. El usuario otorga el comando y espera por el resultado. Ante esto, el tiempo de respuesta debe ser corto (en el orden de segundos como mucho). Un sistema interactivo es utilizado cuando se requiere un tiempo mínimo de respuesta.
Los sistemas operativos de tiempo compartido usan el scheduling de la CPU y la multiprogramación para proveer a cada usuario con una pequeña porción de la computadora de tiempo compartido. Cada usuario tiene por lo menos un programa separado en memoria. Un programa que es cargado en memoria y se ejecuta es normalmente llamado *proceso*. Cuando un proceso es ejecutado, este típicamente se ejecuta por un corto tiempo antes de que finalice o necesite realizar I/O. La I/O puede ser interactiva; es decir, la salida es por medio de un display para un usuario, y la entrada es por medio de un teclado de un usuario. Ya que la I/O interactiva típicamente corre a la velocidad de las personas, este puede tomar un largo tiempo hasta que sea completada. Por ejemplo, la entrada puede estar limitada por la velocidad de tipeo del usuario; 5 caracteres por segundo es bastante rápido para los humanos pero demasiado lento para las computadoras. Esto provoca que la CPU este ociosa cuando toma lugar una de estas entradas interactivas. Ante esto, el sistema operativo rápidamente cambiara la CPU para el programa de algún otro usuario.
Los sistemas operativos de tiempo compartido son incluso más complejos que los sistemas operativos multiprogramados. Como en la multiprogramación muchos trabajos deben mantenerse simultáneamente en memoria, se requiere alguna forma de administración de la memoria y de protección (capítulo 8). Para que se obtenga un tiempo de respuesta razonable, los trabajos se cambian (swap) dentro y fuera de la memoria hacia el disco que ahora sirve como un almacenamiento de apoyo para la memoria principal. Un método común para conseguir este objetivo es el de *memoria virtual*, el cual es una técnica que permite que un trabajo este siendo ejecutado, sin estar éste completamente en memoria (capítulo 9). El mayor beneficio de esta técnica es que los programas pueden ser más grandes que la memoria. Los sistemas de tiempo compartido deben proveer también un sistema de archivo on-line. El sistema de archivos reside en una colección de discos. Aquí también se necesita un mecanismo de administración de los discos. Los sistemas de tiempo compartido también proveen la ejecución concurrente, el cual requiere sofisticados esquemas de scheduling de la CPU.

La multiprogramación y los sistemas de tiempo compartido son los esquemas centrales de los sistemas operativos modernos.

Sistemas de computadoras personales

Como los costos de hardware han decrecido, ha sido posible tener sistemas de computadoras para un único usuario. Estos tipos de sistemas de computadoras son usualmente referidos a computadoras personales (PCs). Los dispositivos de I/O han cambiado: con paneles de cambio y lectores de tarjetas reemplazadas por teclados y mouse. Las impresoras de línea y las tarjetas agujereadas se han reemplazado por pantallas (monitores) y por más pequeñas y más rápidas impresoras. Estos sistemas son microcomputadoras que son considerablemente más pequeñas y menos caras que los sistemas mainframe. Aunque la protección de los archivos puede verse no necesaria en las computadoras personales, estas computadoras son a menudo ligadas a otras computadoras por medio de líneas telefónicas o redes de área local. Cuando otras computadoras y otros usuarios pueden acceder a los archivos de una computadora personal, la protección de los archivos se convierte una vez más en una característica necesaria de un sistema operativo.

Sistemas paralelos

Aunque la mayoría de los sistemas de la fecha tiene un único procesador, es decir, tienen solo una única CPU, existe una tendencia hacia los sistemas multiprocesadores. Tales sistemas tienen más que un procesador en comunicación, compartiendo el bus de la computadora, el reloj, y a veces la memoria y los dispositivos periféricos.

Existen varias razones para la construcción de tales sistemas. Una gran ventaja es la de incrementar el throughput. Al incrementar el número de procesadores, se espera que aumente la cantidad de trabajo realizado en un periodo de tiempo. Los sistemas multiprocesadores pueden también provocar un ahorro de dinero, ya que ellos comparten dispositivos periféricos, etc. Si varios programas operan sobre el mismo conjunto de datos, es más barato tener almacenado dichos datos en un disco y que todos los procesadores compartan dicho disco, en lugar de tener muchas computadoras con discos locales y muchas copias de los datos.

Otra razón para la utilización de los sistemas multiprocesadores es que aumenta la fiabilidad. Si las funciones pueden ser distribuidas apropiadamente entre varios procesadores, entonces la falla de un procesador no parara el sistema, aunque provocara que sea más lento. Ante esto, para poder continuar en caso de que algún procesador falle, se necesita un mecanismo que detecte la falla, realice un diagnostico y que corrija la falla en caso de que sea posible.

Los sistemas Tandem usan la duplicación de hardware y software para asegurar que continúe la operación a pesar de la existencia de fallas. El sistema consiste de dos procesadores idénticos, cada uno con su propia memoria local. Los procesadores están conectados por medio de un bus. Un procesador es el primario y el otro es el backup. En puntos fijos de chequeo en la ejecución del sistema, la información de estado de cada trabajo (incluyendo una copia de la imagen de la memoria) es copiada desde la maquina primaria hacia el backup. Si se detecta un fallo, se activa la copia del backup, y el sistema es restaurado desde el último punto de chequeo. Esta solución es totalmente cara, ya que existe una clara duplicación del hardware.

Los sistemas multiprocesadores actuales usan el modelo de multiprocesamiento simétrico, en el cual cada procesador corre una copia idéntica del sistema operativo, y éstas copias se comunican entre sí como lo necesiten. Algunos sistemas utilizan multiprocesamiento asimétrico, en el cual cada procesador es asignado a tareas especificas. El procesador maestro controla el sistema; los demás procesadores o "miran" al maestro para las instrucciones o tienen tareas predefinidas. Este sistema define una relación maestro-esclavo. El procesador maestro asigna las tareas a los procesadores esclavos.

Sistemas distribuidos

Una tendencia reciente en los sistemas de computadoras es la de distribuir la computación a través de varias computadoras. En contraste con los sistemas paralelos vistos anteriormente, éstos procesadores no comparten memoria o el reloj. En cambio, cada procesador tiene su propia memoria local. Los procesadores

se comunican entre sí a través de varias líneas de comunicación, tales como líneas telefónicas o buses de alta velocidad. Estos sistemas son llamados sistemas distribuidos.
Los procesadores en un sistema distribuido pueden variar en tamaño o función. Ellos pueden incluir pequeños multiprocesadores, estaciones de trabajo, minicomputadoras, y grandes sistemas de computadoras de propósito general. Estos procesadores son referidos por unos números extensos de nombres, tales como, sitios, nodos, computadoras y demás, dependiendo del contexto en el cual se mencionan.
Hay una gran variedad de razones por la cual construir un sistema de tales características. Las más importantes son:

- **Compartir recursos**: Si un número de sitios diferentes (con diferentes capacidades) están conectados entre sí, entonces el usuario de un sitio puede ser capaz de usar los recursos disponibles en otro lugar. Por ejemplo, un usuario de un sitio A puede estar usando una impresora láser que se encuentra disponible solo en otro sitio B. Entre tanto, el usuario de B puede acceder a los archivos que están en A.
- **Aumento de la velocidad de cálculo**: Si una tarea puede ser particionada en varias subtareas que pueden correr concurrentemente, entonces un sistema distribuido nos permitirá realizar el cálculo a través de los varios sitios. Además, si un sitio en particular esta sobrecargado con trabajos, algunos de éstos trabajos pueden ser movidos a otro sitio que esté con pocos trabajos. Este movimiento de trabajos es llamado load sharing.
- **Fiabilidad**: En caso de que un sitio falle en un sistema distribuido, el resto de los sitios pueden continuar la operación.
- **Comunicación**: Hay muchos casos en el cual los programas necesitan intercambiar datos con otro programa en otro sistema. Los sistemas windows son un caso, ya que ellos frecuentemente comparten datos o transfieren datos entre displays. Cuando muchos sitios están conectados entre sí por medio de una red, los procesos de los diferentes sitios tienen la posibilidad de intercambiar información.

Sistemas de tiempo real

Otra forma de sistemas operativos de propósito general es el sistema de tiempo real. Un sistema de tiempo real es utilizado cuando hay requerimientos de tiempo rígidos en la operación de un procesador o el flujo de datos, y así es usado a menudo como un dispositivo de control en una aplicación especial. Los sensores llevan datos a la computadora. La computadora debe analizar los datos y posibilitar el ajuste del sensor. Los sistemas que controlan experimentos científicos, sistemas con imágenes medicas, sistemas de control industrial, y otros son sistemas de tiempo real. Un sistema de tiempo real tiene bien definido la restricción en el tiempo de respuesta. El procesamiento debe ser realizado en el tiempo especificado o el sistema fallara.
Existen dos tipos de sistemas de tiempo real: Un sistema de tiempo real duro garantiza que las tareas críticas se realizaran a tiempo. Este objetivo requiere que todos los retardos en el sistema sean limitados. Ante esto, los sistemas de almacenamiento secundario están generalmente limitados o evitados, estando, en cambio, los datos almacenados en memorias de plazos cortos, o en memoria ROM. Esta ROM es situada en dispositivos de almacenamiento no volátiles, el cual almacenan sus contenidos aun en caso de un corte de la energía. Las más avanzadas características de los sistemas operativos están ausentes también, ya que ellas tienden a separar al usuario más allá del hardware. Por ejemplo, la memoria virtual casi nunca es encontrada en los sistemas de tiempo real. Por consiguiente, los sistemas de tiempo real duros están en conflicto con los sistemas de tiempo compartido, y ambos no pueden ser mezclados.
Un sistema de tiempo real menos restricto es el sistema de tiempo real blando, en donde una tarea critica obtiene prioridad sobre las demás tareas, y retiene la prioridad hasta que ella es completada. Como en los sistemas de tiempo real duro, la demora del kernel necesita ser limitada: una tarea de tiempo real no puede estar esperando indefinidamente por el kernel para que pueda ser ejecutada. Los sistemas de tiempo real blandos necesitan características de sistemas operativos avanzados que no pueden ser soportados por los sistemas de tiempo real duros.

2. Estructura del sistema de la computadora

Operación del sistema de la computadora

Una computadora moderna consiste de una CPU y de un número de controladores de dispositivos conectados a través de un bus común que provee acceso a la memoria compartida (Figura 2.1).

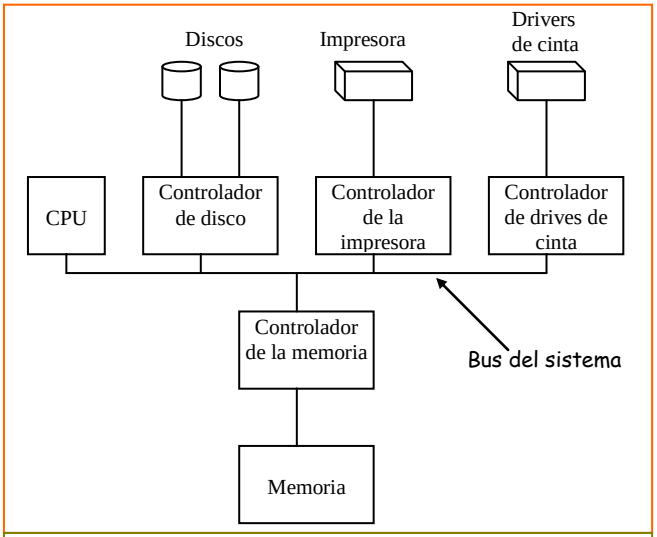


Figura 2.1 Un sistema de computadora moderno.

Cada controlador de dispositivo está a cargo de un determinado tipo de dispositivo (ejemplo, drives de disco, dispositivo de audio y displays de video). Los controladores de dispositivos y la CPU pueden ejecutar concurrentemente compitiendo por los ciclos de la memoria. Para asegurar un acceso ordenado a esta memoria compartida se encuentra el controlador de la memoria el cual tiene la función de sincronizar los accesos a la misma.

Cuando la computadora es iniciada, esta necesita tener un programa inicial para correr. Este programa es llamado *bootstrap program* y su función principal es la de inicializar todos los aspectos del sistema, desde los registros de la CPU hasta los controladores de los dispositivos. Este programa también debe ubicar y cargar en la memoria el sistema operativo. El sistema operativo entonces comienza a ejecutar el primer proceso, sea entonces *init*, y espera ante la ocurrencia de algún evento. La ocurrencia de un evento es provocada por lo que se llama una interrupción desde alguna parte del hardware o software. El hardware puede enviar una interrupción en cualquier momento enviando una señal a la CPU, usualmente por medio del bus del sistema. El software, en cambio, puede enviar una interrupción ejecutando una operación especial llamada *system call*.

Existen muchos tipos de eventos que pueden desencadenar una interrupción (por ejemplo la completitud de una operación de I/O, la división por cero, acceso invalido a memoria, o pedido a algún servicio del sistema operativo). Para cada una de las interrupciones se tiene una rutina de servicio el cual es la responsable de tratar la interrupción.

Cuando la CPU es interrumpida, para lo que este haciendo e inmediatamente transfiere la ejecución a una dirección fija. Esta dirección fija contiene usualmente la dirección de comienzo donde se localiza la rutina de servicio. Así, se ejecuta esta rutina de servicio y al completarse, la CPU continúa lo que estaba haciendo.

La interrupción debe transferir el control a la rutina que servirá la interrupción. Esta dirección debería ser la de comienzo de la rutina de servicio apropiada según el tipo de interrupción. El método más sencillo seria transferir la ejecución a una rutina genérica que examine la información de la interrupción y a partir de esta información derivar la ejecución a la dirección apropiada. Sin embargo, las interrupciones deben ser manejadas rápidamente y ante el hecho que hay un número predefinido de posibles interrupciones, es

más conveniente utilizar una tabla de punteros a las direcciones de las rutinas de interrupciones. La rutina de interrupción es entonces indirectamente llamada a través de la tabla, sin necesitar una rutina intermediaria. Generalmente, esta tabla es almacenada en la parte baja de la memoria (las primeras 100 o más ubicaciones). Estas direcciones almacenan las direcciones de las rutinas de servicio de las interrupciones para los diferentes dispositivos. Esta tabla es llamada *vector de interrupciones*.
La arquitectura de la interrupción debe también almacenar la dirección de la instrucción interrumpida, es decir, la instrucción que la CPU estaba ejecutando justo antes de que "llegue" la interrupción. Los sistemas modernos almacenan esta dirección de retorno en un sistema de pila. Si la rutina de interrupción necesita, por ejemplo, modificar el estado del procesador (modificando por ejemplo el valor de los registros), entonces esta rutina debe explícitamente almacenar el estado actual del procesador y reestablecerlo cuando ya haya utilizado el mismo. Luego de que se trato la interrupción, la dirección de retorno que se había almacenado se vuelve a cargar en el contador del programa y se continúa con el programa como que si la interrupción no hubiera ocurrido.
Los sistemas operativos más modernos son *interrupt driven*. En caso de que no haya procesos para ejecutar, no hay dispositivos de I/O para servir, y no hay usuarios a quienes responder, el sistema operativo estaría en la espera de que algo ocurra. Los eventos son casi siempre señalados por la ocurrencia de una interrupción o trap (o una *exception*). Un trap o exception es una interrupción generada por el software causada ya sea por un error (por ejemplo, la división por cero o el acceso a una dirección invalida de memoria), o por un pedido especifico de un programa de usuario que el sistema operativo debe tratar. En estos tipos de sistemas (es decir, *interrupt driven*), al ocurrir una interrupción (o trap), el hardware transfiere el control al sistema operativo. Primero, el sistema operativo preserva el estado de la CPU almacenando los valores de los registros y el contador del programa. Luego, determina que tipo de interrupción ha ocurrido. Esta determinación puede requerir *polling* (se le pregunta a todos los dispositivos de I/O para detectar quien envió la interrupción), o por medio de un vector de interrupciones. Para cada tipo de interrupción, existen segmentos separados de código en el sistema operativo que determinan que acciones deben ser tomadas.

Estructura de la I/O

Como se dijo anteriormente, cada controlador tiene a cargo un tipo especifico de dispositivo. Pero dependiendo del tipo de controlador, hay controladores que tienen a cargo más de un dispositivo. Por ejemplo, el controlador SCSI (Small Computer Systems Interface, ver figura 12.1), el cual es encontrado en las computadoras de tamaño pequeño o mediano, pueden tener 7 o más dispositivos a cargo. Un controlador tiene un buffer local de almacenamiento y un conjunto de registros de propósito especial. El controlador es el responsable de mover los datos entre los dispositivos que tiene a su control y su buffer local. El tamaño del buffer varia dependiendo del tipo de controlador. Por ejemplo, el tamaño del buffer del controlador de disco es igual o múltiplo del tamaño de la mínima porción direccionable de un disco, el cual es llamado sector, cuyo tamaño es usualmente de 512 bytes.

Interrupciones de I/O: Para comenzar una interrupción de I/O, la CPU carga los registros apropiados en el controlador del dispositivo. El controlador, de hecho, examina el contenido de estos registros para determinar que acción debe tomar. Por ejemplo, si encuentra un pedido de lectura, el controlador comenzara a transferir los datos desde el dispositivo hasta su buffer. Una vez que el dato fue transferido, el controlador le informa a la CPU que ha finalizado su operación. El controlador logra esta comunicación por medio de una interrupción.
Esta situación ocurrirá, en general, como el resultado del pedido de I/O del proceso de un usuario. Una vez que comienza la I/O, dos cursos de acción son posibles. En el caso más simple, se inicia la I/O y al completarse, el control es retornado al proceso del usuario. Este caso es conocido como I/O sincrónica. La otra posibilidad es llamada I/O asincrónica y se retorna el control al programa del usuario sin esperar que se complete la I/O. La I/O puede entonces continuar mientras ocurre otra operación del sistema (Figura 2.3).

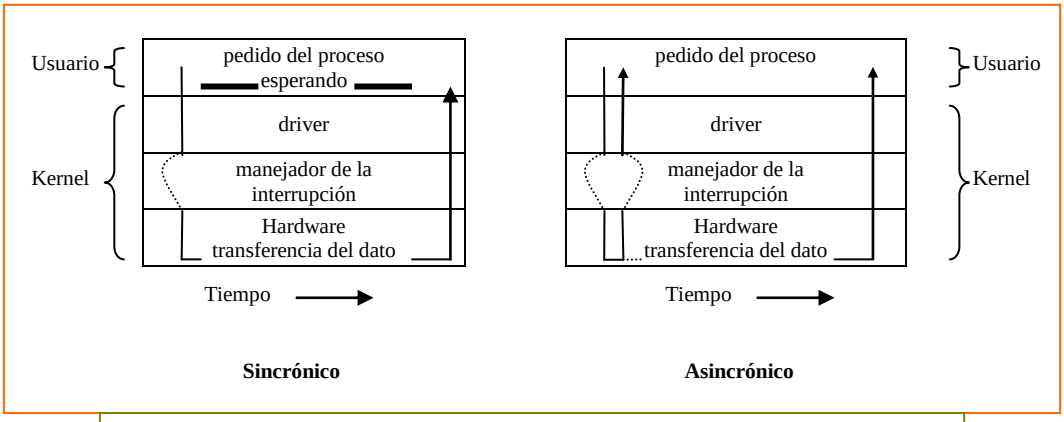


Figura 2.3 Dos métodos de I/O: (a) sincrónico, y (b) asincrónico.

La espera por la completitud de una I/O se puede lograr de dos maneras. Algunas computadoras tienen una instrucción llamada **wait** el cual deja ociosa la CPU hasta la próxima interrupción. Maquinas que no tienen tal instrucción forman un loop especial de espera:

Loop: jmp Loop

Este loop continua hasta que ocurra una interrupción, transfiriendo el control a alguna otra parte del sistema operativo.

En caso de que la CPU siempre espere por la completitud de I/O, a lo sumo un pedido de I/O es atendido en un tiempo dado. Así, cuando ocurre una interrupción de I/O, el sistema operativo conoce exactamente cual es el dispositivo que es interrumpido. Por otro lado, esto excluye concurrentes operaciones de I/O de varios dispositivos.

Una mejor alternativa es la de comenzar una operación de I/O y luego continuar procesando otro código del sistema operativo o del usuario. Un *system call* (o pedido al sistema operativo) se necesita para permitir al programa del usuario que espere por la completitud de la I/O, si es deseado. En caso de que no haya programas del usuario listos para correr, y el sistema operativo no tiene otro trabajo para hacer, provocara que se entre en la instrucción wait o en el loop infinito anteriormente vistos. También se necesita tener registro de todos los pedidos de I/O que se producen en un instante. Para este propósito, el sistema operativo usa una tabla conteniendo una entrada para cada dispositivo de I/O: la tabla de estado de los dispositivos (figura 2.4). Cada entrada de la tabla indica el tipo de dispositivo, la dirección y el estado (no funcionando, ocioso, ocupado). Si el dispositivo esta ocupado con un pedido, el tipo de pedido y otros parámetros serán almacenados en la entrada de la tabla para el dispositivo. Ya que puede que otros procesos realicen pedidos para un mismo dispositivo, el sistema operativo también mantiene una cola de espera para cada dispositivo de I/O.

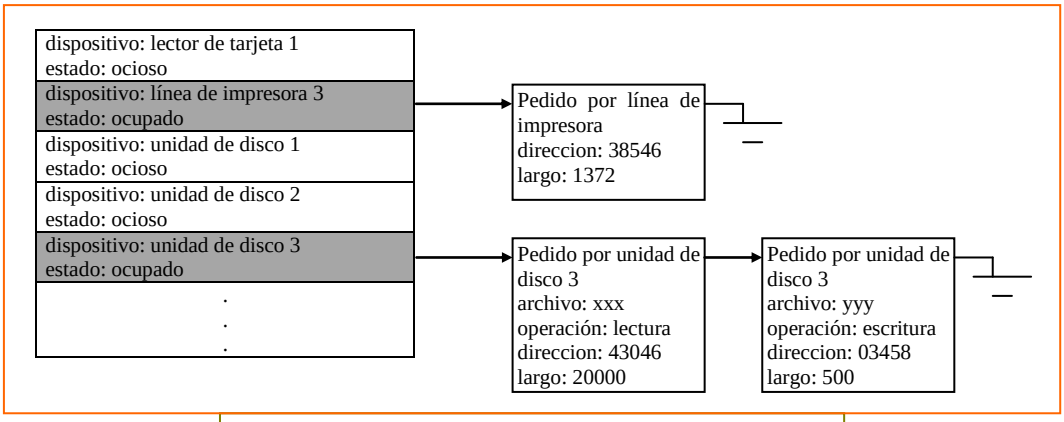


Figura 2.4 Tabla de estado de los dispositivos.

Un dispositivo de I/O interrumpe cuando éste necesita servicio. Al ocurrir una interrupción, el sistema operativo primero determina cual es el dispositivo de I/O que provoco la interrupción. Luego entonces se fija en la tabla cual es el estado de dicho dispositivo y modifica el estado de la entrada para reflejar la ocurrencia de dicha interrupción. Para la mayoría de los dispositivos, una interrupción refleja la completitud de un pedido de I/O. En caso de que haya más pedidos esperando en la cola de espera para dicho dispositivo, entonces el sistema operativo comienza a procesar el siguiente pedido.
Finalmente, el control es retornado desde la interrupción de I/O. Si un proceso estaba esperando para que se complete dicha operación de I/O, ya se le puede retornar el control. De otra forma (es decir, no existía ningún proceso esperando por dicha completitud de I/O), se retorna a lo que se estaba haciendo antes de la interrupción de I/O: a la ejecución de un programa del usuario (el programa comenzó una operación de I/O y la operación no ha finalizado todavía, pero el programa no tiene que esperar todavía por la completitud del pedido), o retorna al loop de espera (el programa comenzó dos o más operaciones de I/O y esta esperando por una operación en particular y la interrupción de completitud fue provocada por alguno de los otros pedidos). En un sistema de tiempo compartido, el sistema operativo cambiaría el control a algún otro proceso que este listo para correr.
La ventaja más importante de la I/O asincrónica es que incrementa la eficiencia del sistema. Mientras la I/O toma lugar, la CPU puede ser usada para procesar o comenzar la I/O de otro dispositivo.

Estructura del DMA: Consideremos un simple driver de entrada de una terminal. Cuando el primer caracter es tipeado se envía a la computadora. Al recibirse dicho caracter, el dispositivo de comunicación asincrónico (o puerto serial), al cual el driver esta conectado, interrumpirá la CPU. Al llegar la interrupción de la terminal, la CPU estará a punto de ejecutar alguna instrucción. En caso de que la CPU este en la mitad de la ejecución de alguna instrucción, la interrupción se mantiene hasta que la instrucción que se está ejecutando finalice. La dirección de esta instrucción interrumpida es almacenada, y el control es transferido a la rutina de servicio de la interrupción para el dispositivo apropiado.
La rutina de servicio almacena el contenido de cualquier registro de la CPU que necesitara usar. Luego, toma el caracter del dispositivo, y almacena dicho caracter en un buffer. La rutina de interrupción debe también ajustar el puntero y las variables, para estar seguro que el siguiente caracter será almacenado en el lugar correcto del buffer. La rutina de interrupción setea luego una bandera en la memoria indicando a las otras partes del sistema operativo que se ha recibido una nueva entrada. Las otras partes son responsables de procesar el dato en el buffer y de transferir el caracter al programa que realizo el pedido de la entrada. Luego, la rutina de servicio de la interrupción restaura el contenido de cualquier registro que anteriormente fue almacenado y transfiere el control a la instrucción que fue interrumpida.
Si los caracteres están siendo tipeados a una velocidad de 9600 baud, la terminal puede aceptar y transferir un caracter en aproximadamente cada 1 milisegundo, o 1000 microsegundos. Como la rutina de servicio de una interrupción para la entrada de un caracter en un buffer puede llevar 2 microsegundos por caracter, deja 998 microsegundos libres de cada 1000 para que los utilice la CPU. Ante esto, la I/O asincrónica es usualmente asignada a las interrupciones de baja prioridad, permitiendo a otras interrupciones más importantes ser procesadas primero. Sin embargo, un dispositivo de alta velocidad (como son los discos, o la comunicación de red) debe ser capaz de transmitir información a la velocidad de la memoria; la CPU necesitaría 2 microsegundos para responder a cada interrupción, y si cada interrupción arriva cada 4 microsegundos (por ejemplo), no le dejaría demasiado tiempo a la CPU para procesar la información.
Para resolver este problema, se utiliza el acceso directo a memoria (*DMA: direct memory access*) para los dispositivos de I/O de alta velocidad. Luego de que se setean los buffers, los punteros y los contadores para el dispositivo de I/O, el controlador del dispositivo transfiere un bloque entero de datos directamente a o desde su propio buffer a la memoria, sin la intervención de la CPU. Solo se genera una interrupción por bloque, en vez de una interrupción por byte (o palabra) generada por los dispositivos de baja velocidad.
La operación básica de la CPU es la misma. Un programa de usuario, o el sistema operativo mismo, puede requerir la transferencia de un dato. El sistema operativo encuentra un buffer (un buffer vacío para la

entrada, o un buffer lleno para la salida) de una pileta de buffers para la transferencia. Un buffer es típicamente de 128 a 4096 bytes, dependiendo del tipo de dispositivo. Luego, una porción del sistema operativo llamado driver del dispositivo (device driver), setea los registros del controlador de DMA para usar la fuente apropiada, la dirección de destino, y el largo a transferir. El controlador del DMA esta entonces listo para comenzar la operación de I/O. Mientras el controlador de DMA esta realizando la transferencia del dato, la CPU esta libre de realizar otras tareas. Ya que la memoria normalmente puede transferir una palabra a la vez, el controlador de DMA "roba" ciclos de memoria de la CPU. Los ciclos robados pueden bajar la ejecución de la CPU, mientras la transferencia esta siendo completada. El controlador del DMA interrumpe a la CPU cuando la transferencia se completo.

Estructura de almacenamiento

Los programas deben estar en la memoria principal para ser ejecutados. La memoria principal es la única área de almacenamiento donde el procesador puede acceder directamente. Esta es un arreglo de palabras o de bytes. Cada palabra tiene su propia dirección. La interacción es lograda a través de una secuencia de instrucciones **load** o **store** a direcciones especificas de la memoria. La instrucción load mueve una palabra desde la memoria principal a un registro interno de la CPU, mientras que la instrucción store mueve el contenido de un registro a la memoria principal. Aparte de las instrucciones load y store explícitas, la CPU automáticamente carga instrucciones desde la memoria para su ejecución.
En un ciclo para la ejecución de una instrucción, siguiendo la arquitectura de Von Neumann, primero se debe cargar una instrucción desde la memoria y almacenar la instrucción en el registro de instrucción (*instruction register*). La instrucción es entonces decodificada y puede causar operandos que deben ser buscados a la memoria y almacenados en algún registro interno. Luego de que la instrucción ha sido ejecutada sobre los operandos, el resultado es almacenado otra vez en la memoria. Note que la unidad de memoria ve solo un stream de direcciones de memoria, ésta no conoce como ellas son generadas (el contador de la instrucción, índices, dirección, y demás) o que hay dentro de ellas (datos o instrucciones).
Idealmente, nosotros deseamos que los datos y el programa este en memoria permanentemente. Esto no es posible por las siguientes dos razones:

- La memoria principal es muy pequeña para almacenar todos los programas y datos que necesita.
- La memoria principal es un dispositivo de almacenamiento volátil que pierde su contenido cuando hay un corte de energía.

Ante esto, la mayoría de las computadoras ofrecen un almacenamiento secundario como una extensión de la memoria principal. El requerimiento más importante que ofrece un almacenamiento secundario es que es capaz de almacenar grandes cantidades de datos permanentemente.
El dispositivo de almacenamiento secundario más común es el disco magnético, el cual provee almacenamiento tanto a datos como a programas. La mayoría de los programas (web browsers, compiladores, procesadores de texto, etc) están almacenados en una unidad de disco y son cargados en memoria.
Pero, las estructuras de almacenamiento que se vieron (consistente de registros, memoria principal y discos magnéticos) es solo una de los muchos sistemas de almacenamiento. También existe la memoria cache, CD-ROM, cintas magnéticas, etc. Sus mayores diferencias están en la velocidad, costo, tamaño y volatilidad.

Memoria principal: La memoria principal y los registros que se encuentran dentro del procesador son los únicos sistemas de almacenamiento en el cual la CPU puede acceder directamente. Por lo tanto, cualquier instrucción en ejecución, y cualquier dato que este siendo usado por la instrucción, debe estar en uno de estos dos dispositivos de almacenamiento de acceso directo. En caso de que el dato no este en memoria, este debe ser cargado allí antes de que la CPU pueda operar sobre él.
En el caso de la I/O, cada controlador de I/O incluye registros que guardan comandos y los datos que esta siendo transferido. Usualmente, instrucciones especiales de I/O permiten la transferencia de datos entre

estos registros y la memoria del sistema. Para permitir un acceso más conveniente a los dispositivos de I/O, muchas arquitecturas de computadoras proveen un mapeo de memoria de I/O. En este caso, los rangos de direcciones de memoria son un conjunto aparte, y son mapeados a los registros del dispositivo. Las lecturas y escrituras de estas direcciones de memoria causan que el dato sea transferido a y desde los registros del dispositivo. Este método es apropiado para aquellos dispositivos con un rápido tiempo de respuesta, tales como los controladores de video. En la PC de IBM, cada lugar de la pantalla es mapeado a un lugar de la memoria.
El mapeo de memoria de I/O es también conveniente para otros dispositivos, tales como puertos seriales o paralelos usados para conectar modems e impresoras a la computadora. La CPU transfiere el dato a través de estos tipos de dispositivos por medio de leer o escribir una pequeña cantidad de los registros del dispositivo llamado puerto de I/O. Para enviar un largo string de bytes a través de un puerto serial de mapeo de memoria, la CPU escribe un byte de dato al registro de dato, luego setea un bit en el registro de control para señalar que el byte esta disponible. El dispositivo toma el byte de dato, y luego limpia el bit en el registro de control para señalar que esta listo para leer el siguiente byte. Así, la CPU puede transferir el siguiente byte. En caso de que la CPU utilice polling para ver el bit de control, entonces realiza un loop constante de ver dicho bit para darse cuenta de cuando el dispositivo esta listo para leer el siguiente byte. Este ultimo método es llamado *programmed I/O* (PIO). Si la CPU en lugar de usar polling recibe una interrupción cuando el dispositivo esta listo, la transferencia del dato se dice que es *interrupt driven*.
Los registros que están dentro de la CPU se pueden acceder en un ciclo de reloj de la CPU. La mayoría de las CPUs pueden decodificar y hacer simples operaciones sobre el contenido de los registros de la CPU al precio de uno o más operaciones por tick de reloj. Pero no se puede decir lo mismo de la memoria principal, el cual es accedida vía una transacción del bus de memoria. El acceso a memoria puede tomar muchos ciclos de reloj para ser completado, en el cual la mayoría de las veces el procesador queda parado por no tener todavía el dato requerido para completar la instrucción que esta siendo ejecutada. Esta situación es intolerable por la frecuencia de accesos a memoria. El remedio es el de agregar una memoria más rápida entre la CPU y la memoria principal. Dicha memoria es llamada *cache* y se vera más adelante (capítulo 2.4.1).

Disco magnético: Los discos magnéticos proveen una gran cantidad de almacenamiento secundario. Conceptualmente los discos son bastantes simples. Cada plato del disco se puede ver como un CD. Las dos caras del disco son cubiertas por un material magnético. Cada cabeza de lectura-escritura recorre cada una de las caras de cada plato. Todas las cabezas están unidas a una brazo, el cual mueve todas las cabezas como una unidad. La cara de un plato es dividida en pistas circulares (tracks), las cuales a su ves se dividen en sectores. Al conjunto de cilindros que se forman a partir de una posición dada del brazo se llaman cilindros. Cuando el disco esta en uso, el motor del drive da vueltas a una gran velocidad. La mayoría de los drives dan desde 60 a 150 vueltas por segundo. La velocidad del disco tiene dos partes. El costo de la transferencia es el costo en el cual el dato fluye entre el drive y la computadora. El tiempo de posicionamiento, a veces llamado *random access time*, consiste del tiempo para mover el brazo del disco hacia el cilindro deseado, llamado el tiempo de posicionamiento, y el tiempo para ubicarse en el sector deseado, el cual se debe rotar el disco hasta la cabeza lectora, llamado tiempo de latencia rotacional. Los discos típicos pueden transferir varios megabytes de datos por segundo, y tienen tiempo de ubicación y latencia rotacional de varios milisegundos.
A veces la cabeza lectora hace contacto con la cara del disco. Aunque los platos del disco están cubiertos por una pequeña protección, a veces la cabeza daña esta protección. Esto es llamado *head crash*, y tiene como consecuencia que el disco no pueda ser reparado y debe ser totalmente reemplazado.
El drive del disco esta unido a la computadora por medio de un conjunto de cables llamado bus de I/O. La transferencia de los datos sobre un bus es llevada a cabo por procesadores electrónicos especiales llamados controladores. El *host controller* es el controlador de la computadora que esta al final del bus. Un controlador del disco (*disk controller*) esta construido en cada drive de disco. Para realizar una operación de I/O de disco, la computadora ubica un comando en el host controller, típicamente usando puertos de I/O de mapeo de memoria. Luego, el host controller envía el comando al controlador de disco, y el

controlador de disco opera sobre el hardware del drive del disco para llevar a cabo el comando. Los controladores de disco tienen usualmente dentro de ellos construidos una cache. La transferencia del dato al drive del disco ocurre entre la cache y la cara del disco, y la transferencia del dato al host, a gran velocidad electrónica, ocurre entre la cache y el host controller.

Cinta magnética: las cintas magnéticas fueron usadas antiguamente como un medio de almacenamiento secundario. Aunque es relativamente permanente y puede almacenar grandes cantidades de datos, su tiempo de acceso es muy bajo en comparación con los discos magnéticos. En este momento, las cintas son usadas principalmente para backups, para almacenar información que no es usada frecuentemente, y como un medio de transferir información de un medio a otro.

Jerarquía de almacenamiento

La gran variedad de los sistemas de almacenamiento de una computadora se puede organizar en una jerarquía acorde a su velocidad y costo. Los niveles más altos son caros, pero rápidos. A medida que nos movemos hacia debajo en la jerarquía, el costo por bit decrece, pero el tiempo de acceso generalmente se incrementa.

Además de la velocidad y el costo, también existe el punto de la volatilidad. El almacenamiento volátil pierde su contenido cuando se elimina la energía. En la figura 2.6, los sistemas de almacenamiento que se encuentran encima de los discos son volátiles, mientras que los que están debajo del disco electrónico son no volátiles. Un disco electrónico se puede diseñar para que sea volátil o no-volátil. Durante la operación normal, el disco electrónico almacenan los datos en un largo arreglo DRAM, el cual es volátil. Pero muchos dispositivos de discos electrónicos contienen un disco duro magnético oculto y una batería como reserva de energía. En caso de que se produzca un corte de energía, el controlador de disco electrónico copia los datos desde la RAM al disco magnético. Cuando se produce la restauración de la energía, el controlador copia el dato nuevamente en la RAM.

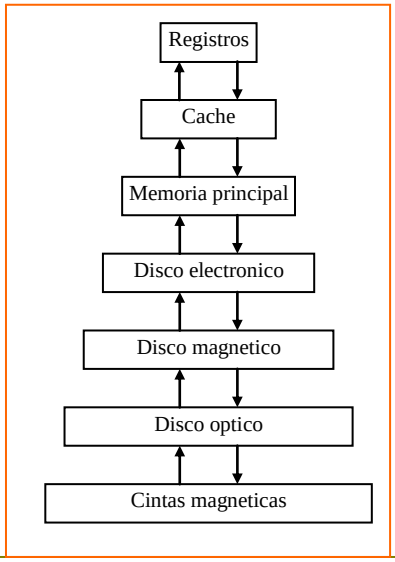


Figura 2.6 Jerarquía de los dispositivos de almacenamiento

Caching: la idea de caching sostiene lo siguiente: la información es normalmente mantenida en algún sistema de almacenamiento (tal como memoria principal). Cuando esta necesita ser usada, se copia en un sistema de almacenamiento más rápido (la cache) sobre términos temporarios (es decir, se copia en la cache cuando se necesita con la idea que será nuevamente utilizada en un corto tiempo. Si pasa un tiempo y no se volvió a usar seguramente sea reemplazada por otra información). Cuando se necesita una pieza particular de información, primero se chequea para ver si dicha pieza esta en la cache. En caso de que este, la información se usa directamente de la cache; si no esta, la información se usa desde el sistema de

almacenamiento principal (memoria principal), poniendo una copia en la cache bajo la suposición de que dicha pieza será próximamente usada.
Extendiendo esta idea, los registros programables internos serían una cache de alta velocidad para la memoria principal. El programador (o el compilador) implementa algoritmos para la asignación de registros y la re ubicación de registros para decidir que información es mantenida en estos registros y que información se mantiene en la memoria principal. Existen también caches que se implementan totalmente en hardware. Por ejemplo, la mayoría de los sistemas tienen una cache de instrucción el cual lo que hace es almacenar la siguiente instrucción a ejecutar. Sin esta cache, la CPU debería esperar varios ciclos de reloj mientras la instrucción es traía desde la memoria principal. Por razones similares, la mayoría de los sistemas tienen uno o más caches de datos de alta velocidad en la jerarquía de memoria.
Ante el hecho que los tamaños de las caches es limitado, el administrador de la cache es muy importante.
La memoria principal se puede ver como una cache rápida de la memoria secundaria, ya que los datos en el almacenamientos secundario se deben copiar en la memoria para su uso, y el dato debe estar en la memoria principal antes de que sea movido nuevamente al almacenamiento secundario. Un dato en el sistema de archivo puede aparecer en varios niveles de la jerarquía. En el nivel más alto, el sistema operativo puede mantener una cache de datos del sistema de archivos en la memoria principal. El almacenamiento secundario más usado es el disco magnético, el cual a menudo las cintas magnéticas o los discos removibles sirven como backups de estos discos magnéticos (también llamados discos duros).
El movimiento de datos entre los diferentes niveles de la jerarquía puede ser explícito o implícito, dependiendo del diseño del hardware y del sistema operativo que controla el sistema. Por ejemplo, los datos transportados desde la cache a la CPU y los registros es usualmente una función del hardware, sin la intervención del sistema operativo. Por otro lado, la transferencia de los datos desde el disco a la memoria es usualmente controlada por el sistema operativo.

Coherencia y consistencia: como se dijo, en la jerarquía un mismo dato puede aparecer en varios niveles. Por ejemplo, consideremos un entero A ubicado en el archivo B el cual esta para ser incrementado por 1. Supongamos que el archivo B reside en un disco magnético. La operación de incremento es precedida primero por la emisión de una instrucción de carga del bloque donde se encuentra A desde el disco a la memoria principal. Este paso es seguido por la posible copia de A en la cache, y luego copiando A en alguno de los registros internos. Así, se ve que la copia de A aparece en varios lugares. Una vez que se produjo el incremento de A en los registros, el valor de A difiere según en el nivel de la jerarquía que nos encontremos. Solo el valor de A será el mismo en todos los niveles cuando se copie nuevamente dicho entero en el disco magnético.
En un entorno donde existe solo un proceso ejecutando a la vez, esto no provoca problema ya que un acceso a A siempre se obtendrá la copia del nivel más alto de la jerarquía. Sin embargo, en un entorno multitarea, donde la CPU esta ejecutando varios procesos "a la vez", se debe tener mucho cuidado para asegurar que, si varios procesos desean acceder a A, entonces cada uno de estos procesos obtendrá el cambio más reciente del valor de A.
Esta situación se transforma más complicada en un ambiente multiproceso donde, además de mantener registros internos, la CPU también contiene una cache local. En tal ambiente, una copia de A puede existir simultáneamente en varias caches. Ya que las varias CPUs pueden todas ejecutar concurrentemente, debemos estar seguros que un cambio en el valor de A se refleje en todas las demás caches que contiene A. Este problema es llamado cache choerency, y es una cuestión de hardware (manejado debajo del nivel del sistema operativo).
En un entorno distribuido, la situación se convierte aun más compleja. En tales ambientes, varias copias de un mismo archivo pueden estar en diferentes computadoras que están distribuidas en el espacio. Ya que las varias réplicas pueden ser accedidas y modificadas concurrentemente, se debe estar seguro que cuando una replica es modificada, las demás vean también dicho cambio lo más pronto posible.

Protección del hardware

Un sistema operativo bueno debe asegurar que un incorrecto (o malicioso) programa no cause que otros programas sean ejecutados incorrectamente.
Muchos errores de los programas son detectados por el hardware. Estos errores son manejados por el sistema operativo. En caso de que un programa del usuario falle en alguna forma (ya sea por ejecutar una instrucción ilegal, o el acceso a memoria en una dirección que no es del espacio del usuario), entonces el hardware enviara una excepción al sistema operativo. La excepción transfiere el control al vector de interrupciones del sistema operativo, así como lo hacia una interrupción. Cada vez que ocurre un error en un programa, el sistema operativo debe terminar el programa de una manera anormal. Un mensaje de error apropiado se presenta y la memoria del programa se libera.

Operación en modo dual: Para asegurar una operación apropiada, debemos proteger al sistema operativo y a los demás programas y sus datos de cualquier programa que funcione mal. La protección es necesaria para cualquier recurso compartido. El idea es la de proveer hardware que nos permita diferenciar entre varios modos de ejecución. Existen dos tipos de modos de ejecución: el modo usuario y el modo monitor. Un bit, llamado el bit de modo, se agrega al hardware de la computadora para indicar el modo actual: monitor (0) o usuario (1). Con el bit de modo, se es capaz de distinguir entre una ejecución que es hecha por el sistema operativo, y otra que es hecha por el usuario.
Al inicio, el hardware comienza en modo monitor. El sistema operativo es entonces cargado, y comienza el usuario a procesar en modo usuario. Al ocurrir una interrupción o una excepción, el hardware se cambia de modo usuario a modo monitor (es decir, cambia el bit de modo a 0). Así, cuando el sistema operativo gana el control de la computadora, ésta esta en modo monitor. El sistema siempre cambia a modo usuario (seteando el bit de modo a 1) antes de pasar el control a un programa de usuario.
El modo dual de operación provee la protección necesaria al sistema operativo de usuarios errantes. Esta protección se logra designando algunas de las instrucciones de maquina que pueden causar daño como *instrucciones privilegiadas*. El hardware permite instrucciones privilegiadas para ser ejecutadas solo en modo monitor. En caso de que se llegue al momento de querer ejecutar una instrucción privilegiada en modo usuario, el hardware no ejecutara la instrucción, tratara la instrucción como una instrucción ilegal y provocara una excepción al sistema operativo.
La falta de hardware que no soporte modo dual puede causar varios defectos en el sistema operativo. Por ejemplo, MS-DOS fue escrito sin el bit de modo, por lo que no tienen modo dual. Un programa de usuario que estaba corriendo en un lugar mal colocado podía llegar a borrar el sistema operativo escribiendo datos sobre éste, y múltiples programas eran capaces de escribir para un dispositivo al mismo tiempo provocando resultados desastrosos.

Protección de I/O: Un programa de usuario puede interrumpir el normal funcionamiento del sistema emitiendo instrucciones ilegales de I/O, o accediendo a lugares de memoria donde se encuentra el sistema operativo, o el rechazo de abandonar la CPU. Podemos utilizar varios mecanismos para asegurar que tales situaciones no pueden tomar lugar en el sistema.
Para asegurar que un usuario no realice instrucciones ilegales de I/O, se definen que todas las instrucciones de I/O son privilegiadas. Así, los usuarios no pueden emitir instrucciones de I/O directamente, éstas deben ser realizadas a través del sistema operativo. Para que la protección de la I/O sea completa debemos asegurarnos que el usuario nunca pueda ganar el control de la computadora en modo monitor. Si esto ocurre, la protección de la I/O podría estar comprometida.
Consideremos que la computadora esta ejecutando en modo usuario. Esta pasara a modo monitor cuando ocurra una interrupción o excepción, saltando a una dirección determinada del vector de interrupciones. Supongamos que el programa del usuario, como parte de su ejecución, almacena una nueva dirección en el vector de interrupciones. Esta nueva dirección podría sobreescribir una dirección previa con una dirección en el programa del usuario. Entonces, cuando ocurra una interrupción o una excepción, el hardware pasara

a modo monitor y transferiría el control (por medio de la nueva modificación) a un programa de usuario. Así, el programa del usuario ganaría el control de la computadora en modo monitor.

Protección de la memoria: Para asegurar una correcta operación, se debe proteger al vector de interrupción de modificación de programas del usuario. Además, debemos también proteger las rutinas de servicio de interrupciones en el sistema operativo de modificaciones. Ante esto, debemos proveer una protección de memoria por lo menos para el vector de interrupciones y para las rutinas de servicio de interrupciones del sistema operativo. En general, sin embargo, lo que se quiere es proteger al sistema operativo de los programas de los usuarios, y también, proteger un programa de usuario de algún otro programa de usuario. Esta protección es proveída por el hardware y puede ser implementada de varias maneras. Aquí veremos una de ellas.
Lo que necesitamos para separar el espacio de memoria de cada programa es determinar el rango de direcciones legales en las que el programa puede acceder. Podemos proveer esta protección utilizando dos registros, usualmente llamados *base* y *límite* (figura 2.7). El registro base contiene la dirección de memoria física legal más pequeña; el registro limite contiene el tamaño del rango. Por ejemplo, si el registro base contiene 300040 y el registro limite es 120900, entonces el programa puede acceder legalmente a todas las direcciones desde 300040 hasta 420940 inclusive.

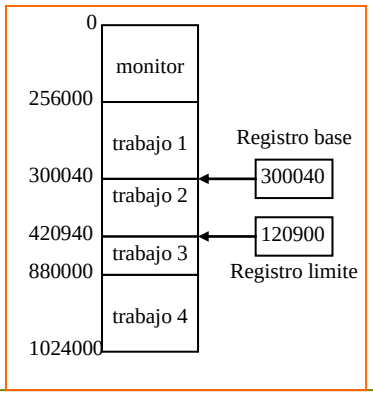


Figura 2.7 Un registro base y un registro límite definen un espacio de direcciones lógicas.

La protección se logra por el hardware de la CPU comparando cada dirección que se logra en modo usuario con los registros. Cualquier intento de un programa ejecutando en modo usuario de acceder a la memoria monitor o a la memoria de otros usuarios resulta en un trap al monitor, el cual trata al intento como un error fatal (figura 2.8). Los registros base y limite son cargados sólo por el sistema operativo el cual utiliza instrucciones privilegiadas especiales. Ya que las instrucciones privilegiadas pueden ser ejecutadas solo en modo monitor, y ya que solo el sistema operativo ejecuta en modo monitor, solo el sistema operativo puede cargar los registros base y limite. Cuando el sistema operativo ejecuta en modo monitor, tiene acceso irrestricto tanto a la memoria monitor como a la memoria del usuario. Esto permite al sistema operativo cargar programas del usuario en la memoria del usuario, eliminar de la memoria dichos programas en caso de error, o acceder y modificar parámetros de las llamadas al sistema (system calls), etc.

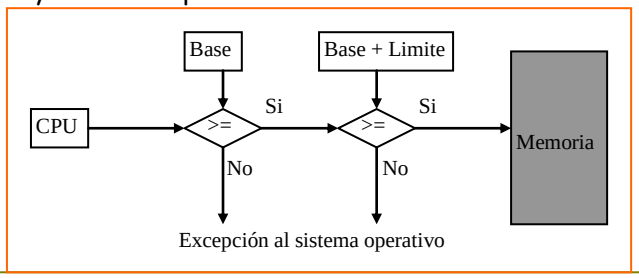


Figura 2.8 Protección de las direcciones con los registros base y límite.

Protección de la CPU: La tercer pieza en el tema de la protección es el de prevenir que los programas de los usuarios caigan en un loop infinito, y nunca retornen el control al sistema operativo. Para lograr esta protección podemos usar un timer, el cual se puede setear para interrumpir la ejecución a períodos de tiempo especificados. El sistema operativo setea el contador. Cada vez que se cumple un tic de reloj, se decrementa el contador. Cuando el contador llega a 0 se produce una interrupción.
Antes de que retorne el control al usuario, el sistema operativo se asegura que el timer esté seteado para interrumpir. En caso de que el timer interrumpa, el control se transfiere automáticamente al sistema operativo, el cual puede tratar la interrupción como un error fatal, o puede otorgar al programa más tiempo. Instrucciones que modifican la operación del timer son claramente privilegiadas.
El uso más común del timer es el de implementar tiempo compartido. En el caso más sencillo, el timer podría ser seteado para interrumpir cada N milisegundos, donde N es la rodaja de tiempo que se le asigna a cada usuario para ejecutar antes de que el siguiente usuario consiga el control de la CPU. El sistema operativo es invocado al final de cada rodaja de tiempo para llevar a cabo varias tareas, tales como resetear registros, variables internas, y buffers, y cambiar varios otros parámetros para preparar para correr el siguiente programa (este procedimiento se conoce como *context switch*). Siguiendo con el context switch, el siguiente programa continua su ejecución desde el punto en el cual fue desalojado (en su anterior rodaja de tiempo).

Arquitectura general del sistema

Ante el desarrollo de sistemas multi-prograd os y de sistemas compartidos, donde los recursos de la computadora se comparten entre los diferentes programas y procesos, se debió modificar la arquitectura básica de la computadora, para permitir al sistema operativo mantener el control del sistema de la computadora y de la I/O.
Para mantener el control, los diseñadores introdujeron un modo dual de ejecución (modo usuario y modo monitor). Este esquema soporta el concepto de instrucciones privilegiadas, el cual puede ser ejecutada solo en modo monitor. Las instrucciones de I/O y las instrucciones para modificar los registros de administración de la memoria (base y limite) o el timer son instrucciones privilegiadas.
Como se imagina, varias otras instrucciones son del tipo privilegiadas. Por ejemplo, la instrucción **halt** es privilegiada. Un programa de usuario nunca debe ser capaz de detener la computadora. Las instrucciones para entrar y salir del sistema de interrupciones son también privilegiadas. La instrucción para cambiar de modo usuario a modo monitor es también privilegiada, y en muchas maquinas, cualquier cambio al bit de modo es privilegiada.
Ya que las instrucciones de I/O son privilegiadas, solo pueden ser ejecutadas por el sistema operativo. La pregunta seria como es que el usuario realiza operaciones de I/O. La solución es que el usuario realice estas operaciones privilegiadas realizando preguntas al monitor para realizar I/O en nombre del usuario.
Tales pedidos son conocidos como llamados al sistema (system call). Un system call es invocado en una gran variedad de formas, dependiendo del funcionamiento proveído por el procesador. De todas formas, éste método es usado por un proceso para pedir una acción al sistema operativo. Un system call usualmente toma la forma de una trap a un lugar especifico en el vector de interrupciones. Este trap puede ser ejecutada por una instrucción genérica **trap**, aunque algunos sistemas (tales como la familia MIPS R2000) tienen una instrucción especifica **syscall**.
Cuando se ejecuta un system call, ésta es tratada por el hardware como una interrupción del software. El control pasa a través del vector de interrupciones a la rutina de servicio del sistema operativo, y el bit de modo es seteado a modo monitor. La rutina de servicio de un system call es una parte de sistema operativo. El monitor examina la instrucción que interrumpió para determinar que system call ha ocurrido; los parámetros indican que tipo de servicio es el que requiere el programa del usuario. Información adicional se necesita para pasar direcciones de memoria e información del llamado. El monitor verifica que los parámetros sean correctos y legales, ejecuta el pedido, y retorna el control a la instrucción que le seguía al system call.

3. Estructuras del sistema operativo

Componentes del sistema

Se puede diseñar un sistema tan grande y complejo como un sistema operativo solo particionando éste en pequeñas piezas. Cada una de estas piezas será una porción bien definida del sistema, con entradas, salidas y funciones cuidadosamente definidas. Obviamente no todos los sistemas tienen la misma estructura. Sin embargo, muchos sistemas modernos comparten el objetivo de soportar los tipos de componentes de sistema que se verán a continuación.

Administrador de procesos: Un programa no es nada a menos que sus instrucciones sean ejecutadas por la CPU. Se podría decir que un proceso es un programa en ejecución, pero ésta es una definición muy acotada. Un trabajo cargado es un proceso. El timer compartido del programa de un usuario es un proceso. Una tarea del sistema es un proceso, tal como enviar información a una impresora. Como veremos, existen system calls que permiten a procesos crear subprocesos para ejecutar concurrentemente.
Un proceso necesita ciertos recursos, incluyendo tiempo de CPU, memoria, archivos, y dispositivos de I/O para lograr su tarea. Estos recursos son dados al proceso cuando éste es creado, o se les van asignando a medida que se va ejecutando. Además de varios recursos lógicos y físicos que un proceso obtiene cuando es creado, algunos datos de inicialización pueden ser pasados. Por ejemplo, consideremos un proceso en el cual su misión es la de mostrar los datos de un archivo sobre la pantalla o sobre alguna terminal. Al proceso se le dará como entrada el nombre del archivo y ejecutara las instrucciones apropiadas y system call para obtener la información deseada y mostrarla en la terminal. Cuando el proceso termina, el sistema operativo reclamara cualquier recurso reusable.
Se debe tener en claro que un programa por sí solo no es un proceso. Un programa es una entidad pasiva, tal como el contenido de algún archivo almacenado en disco, mientras que un proceso es una entidad activa, con un contador de programa especificando la siguiente instrucción a ejecutar. La ejecución de un proceso debe progresar en orden secuencial. La CPU ejecuta una instrucción del proceso, luego otra y así hasta que el programa se complete. Además, en cualquier punto, como mucho una instrucción es ejecutada por un proceso. Así, aunque dos procesos puedan ser asociados con el mismo programa, ellos son sin embargo considerados dos secuencias de ejecuciones separadas. Es muy común tener un programa que produce muchos procesos a medida que se va ejecutando
Un proceso es la unidad de trabajo en un sistema. Un sistema consiste de una colección de procesos, algunos son procesos del sistema operativo (aquellos que ejecutan código del sistema) y el resto son procesos del usuario (aquellos que ejecutan código del usuario).
El sistema operativo es responsable por las siguientes actividades en conexión con la administración de procesos:

- La creación y eliminación tanto de procesos del usuario como del sistema.
- La suspensión y continuación de procesos.
- La provisión de mecanismos para la sincronización de procesos.
- La provisión de mecanismos para la comunicación de procesos.
- La provisión de mecanismos para el manejo de deadlock.

Administración de la memoria principal: La memoria principal es un largo arreglo de palabras o bytes. Cada palabra o byte tiene su propia dirección. La memoria principal es un repositorio de datos de acceso rápido compartido por la CPU y los dispositivos de I/O. El procesador lee instrucciones de la memoria principal durante la etapa de búsqueda de la instrucción, y lee o escribe datos en la parte de ejecución de la instrucción. Las operaciones de I/O implementadas vía DMA también leen o escriben datos en la memoria principal.
Para que un programa pueda ser ejecutado, primero debe ser mapeada la dirección y cargarse el programa en la memoria principal. Mientras el programa se esta ejecutando, se va accediendo a las instrucciones del

mismo y a sus datos desde la memoria por medio de la generación de las direcciones. Cuando el programa termina, el espacio de memoria que éste estaba ocupando se declara disponible, y el siguiente programa puede ser cargado y ejecutado.
Para incrementar tanto la utilización de la CPU y la velocidad de respuesta de la computadora a los usuarios, se deben mantener varios programas en memoria. Existen varios esquemas diferentes de administración de la memoria. La elección de un esquema u otro depende de muchos factores, especialmente del diseño de hardware del sistema. Cada algoritmo requiere su propio soporte de hardware par que pueda ser implementado.
El sistema operativo es responsable de las siguientes actividades en conexión con la administración de la memoria:

- Mantener un conocimiento de que partes de la memoria están siendo usadas y por quien.
- Decidir que procesos están para ser cargados en memoria en el momento que un espacio esta disponible.
- Asignar y des-asignar espacio de memoria como lo necesite.

Administración de archivos: La administración de archivos es una de las partes más visibles de un sistema operativo. Las computadoras pueden almacenar información en varios tipos de medios físicos. Las cintas magnéticas, los discos magnéticos, y los discos ópticos son los medios más comunes. Cada uno de ellos tiene sus propias características y organizaciones físicas. Cada uno es controlado por un dispositivo, tal como el drive de disco o el drive de cinta, con sus propias características. Estas características incluyen su velocidad, capacidad, costo de transferencia de datos, y método de acceso (secuencial o aleatorio).
El sistema operativo mapea archivos sobre el medio físico, y accede a estos archivos vía los dispositivos de almacenamiento. Los archivos son generalmente organizados en directorios para hacer más fácil su uso. Finalmente, cuando varios usuarios tienen acceso a los mismos archivos, es deseable que se controle por quien y de que formas se puede acceder a un archivo dado.
El sistema operativo es responsable por las siguientes actividades acordes a la administración de archivos:

- La creación y eliminación de archivos.
- La creación y eliminación de directorios.
- Soportar primitivas para la manipulación de archivos y directorios.
- El mapeo de archivos en el almacenamiento secundario.
- El bachup de archivos en medio de almacenamiento estable (no volátil).

Administración del sistema de I/O: Una de los propósitos del sistema operativo es el de ocultar las peculiaridades de los dispositivos de hardware específicos al usuario. Por ejemplo, en UNIX las peculiaridades de los dispositivos de I/O son escondidas por el *subsistema de I/O*. Este subsistema consiste de:

- Un componente de administración de memoria incluyendo buffering, caching y spooling.
- Una interfase general del driver del dispositivo.
- Drivers para los dispositivos de hardware específicos.

Solo el driver del dispositivo conocen las peculiaridades del dispositivo especifico para el cual este es asignado.

Administración del almacenamiento secundario: El propósito más importante de una computadora es el de ejecutar programas. Estos programas, con los datos que ellos acceden, deben estar en la memoria principal (almacenamiento primario) durante la ejecución. Ya que la memoria principal es demasiada chica para dar lugar a todos los datos y programas, y estos datos se pierden cuando la energía es cortada, la computadora debe proveer un almacenamiento secundario que sirva como copia de la memoria principal. La mayoría de los programas (incluyendo compiladores, assemblers, rutinas de ordenación, editores y formateadores) son almacenados en el disco hasta que sean cargados en la memoria, y luego utilizan el disco tanto como fuente y como destino de sus procedimientos.

El sistema operativo es responsable por las siguientes actividades en conexión a la administración de discos:

- Administración del espacio libre.
- Asignar almacenamiento.
- Scheduling de disco.

Ya que los sistemas de almacenamiento secundario son utilizados frecuentemente, estos deben ser eficientes.

Networking: Un sistema distribuido es una colección de procesadores el cual no comparten memoria, dispositivos periféricos o el reloj. En lugar de ello, cada procesador tiene su propia memoria local, reloj, y los procesadores se comunican entre sí a través de varias líneas de comunicación, tal como buses de alta velocidad o líneas telefónicas. Los procesadores en un sistema distribuido varían en tamaño y función. Ellos pueden incluir pequeños procesadores, estaciones de trabajo, minicomputadoras, y grandes sistemas de computadoras de propósito general.
Los procesadores en el sistema están conectados por una comunicación vía red, el cual puede ser configurada en un número diferente de formas. La red puede ser parcialmente o totalmente conectada. El diseño de comunicación de red debe considerar el ruteo y estrategias de conexión, y los problemas de contención y seguridad.
Un sistema distribuido posibilita que varios sistemas heterogéneos se unan en un único sistema, dando la posibilidad al usuario de acceder a los varios recursos que el sistema mantiene. El acceso a los recursos compartidos logra velocidad de calculo e incrementa los datos disponibles.

Sistema de protección: si un sistema de computadora tiene múltiples usuarios y permite la ejecución concurrente de múltiple procesos, entonces los diferentes procesos deben ser protegidos de las actividades de algún otro. Para tal propósito, hay mecanismos que aseguran que los archivos, los segmentos de memoria, la CPU y otros recursos, pueden ser operados solo por aquellos procesos que tienen la autorización del sistema operativo.
Por ejemplo, el hardware de direccionamiento de memoria asegura que un proceso puede ejecutar en su propio espacio de memoria. El timer asegura que no puede haber procesos que ganen el control de la CPU sin su eventual renuncia. Finalmente, los registros de control de dispositivos no pueden ser accedidos por los usuarios, por lo que se protege la totalidad de los dispositivos periféricos.

Sistema del Interprete de Comandos: Uno de los programas más importantes del sistema es el interprete de comandos, el cual es la interface entre el usuario y el sistema operativo. Algunos sistemas operativos incluyen el intérprete de comandos en el kernel. Otros sistemas operativos, tal como MS-DOS y UNIX, tratan al intérprete de comandos como un programa especial que esta corriendo cuando un trabajo es iniciado, o cuando el primer usuario entra al sistema (en un sistema de tiempo compartido).
Muchos comandos son otorgados por el sistema operativo por medio de sentencias de control. Cuando en un sistema de lotes comienza un nuevo trabajo, o cuando un usuario entra al sistema en un sistema de tiempo compartido, un programa que lee e interpreta las sentencias de control se ejecuta automáticamente. Este programa es el *interprete de comandos* y se lo conoce como *shell*. Su función es muy simple: conseguir el siguiente comando y ejecutarlo.
Cuando un usuario ingresa al sistema, se inicia un shell. El shell tiene a la terminal como entrada y salida estándar. Muchos sistemas operativos son diferentes en el shell. Un interprete de comandos amigable con el usuario hace que el sistema sea más agradable. Un estilo de interface amigable es el compuesto por ventanas y el mouse, y un sistema de menú (Microsoft Windows). El mouse se mueve para ubicar el puntero sobre las imágenes (iconos) sobre la pantalla que representa los programas, archivos y funciones del sistema. Dependiendo de la ubicación del puntero del mouse, hacer click en un botón del mouse puede invocar un programa, seleccionar un archivo o directorio (conocida como folder o carpeta), o cerrar un menú que contenía comandos. Otros shells más poderosos, complejos y difíciles para aprender son

apreciados por otros usuarios. En algunos de estos shells, los comandos son tipeados sobre un teclado y mostrados en un monitor o impresora, con la tecla *enter* señalando que el comando se finalizo de escribir y esta listo para ser ejecutados. Como éste son los shells de UNIX y MS-DOS.

Servicios del sistema operativo

Un sistema operativo provee un entorno para la ejecución de programas. El sistema operativo provee servicios a los programas y a los usuarios de estos programas. Existen servicios que son comunes para todos los tipos de sistemas operativos, los cuales son:

- *Ejecución de programas:* El sistema debe ser capaz de cargar un programa en la memoria y ejecutarlo. El programa debe ser capaz de finalizar su tarea ya sea de manera normal o anormal (indicando error).
- *Operaciones de I/O:* Un programa corriendo puede requerir un servicio de I/O. Este servicio puede implicar un archivo o un dispositivo de I/O. Por eficiencia y protección, los usuarios usualmente no pueden controlar los dispositivos de I/O directamente. Ante esto, el sistema operativo debe proveer mecanismos para lograr la I/O.
- *Manipulación del sistema de archivos:* Es común que los programas lean y escriban, creen y eliminen archivos.
- *Comunicaciones:* Existen muchas circunstancias en el cual un proceso necesite intercambiar información con otro. Hay dos formas principales en el cual se puede lograr dicha comunicación. La primera toma lugar entre procesos que se están ejecutando en la misma computadora; la segunda toma lugar entre procesos que están en diferentes sistemas de computadoras el cual están unidos por una comunicación de red. La comunicación puede ser implementada vía *memoria compartida*, o por la técnica de *traspaso de mensajes*, en el cual los paquetes de información se mueven entre los procesos por el sistema operativo.
- *Detección de errores:* El sistema operativo constantemente necesita estar alerta de posibles errores. Los errores pueden ocurrir en la CPU y en el hardware de la memoria (tal como un error de memoria o fallo de energía), en dispositivos de I/O (tal como error de paridad en cintas, una conexión de red que fallo, falta de papel en la impresora), o en los programas de los usuarios (tal como un overflow aritmético, intento de acceder a una dirección ilegal de memoria, o demasiado tiempo de uso de la CPU). Para cada tipo de error, el sistema operativo debe realizar las acciones apropiadas para asegurar el correcto y consistente funcionamiento.

Además, existe otro conjunto de funciones del sistema operativo no para ayudar al usuario, sino para asegurar el correcto funcionamiento del sistema. Los sistemas con múltiples usuarios ganan eficiencia compartiendo los recursos de la computadora a través de los diferentes usuarios:

- *Asignación de recursos:* cuando hay múltiples usuarios y múltiples trabajos corriendo al mismo tiempo, los recursos deben ser asignados a cada uno de ellos. Muchos tipos de recursos diferentes son administrados por el sistema operativo. Algunos (tales como los ciclos de la CPU, la memoria principal, y el almacenamiento de archivos) pueden tener código de asignación especial, mientras que otros (tales como los dispositivos de I/O), pueden tener muchos más pedidos generales y código más libre. Por ejemplo, en la determinación de cómo utilizar más eficientemente la CPU, el sistema operativo tiene rutinas de scheduling de la CPU, el cual tiene en cuenta la velocidad de la CPU, los trabajos que deben ser ejecutados, el número de registros disponibles, y otros factores.
- *Informe:* Es deseable mantener un informe de cuales son los usuarios que están usando los recursos y que recurso utiliza cada usuario. Este informe puede servir para estadísticas y por medio del estudio de estas estadísticas se puede llegar a reconfigurar el sistema para aumentar su eficiencia.
- *Protección:* El dueño de la información almacenada en un sistema multiusuario puede desear de controlar su uso. Cuando varios procesos ejecutan concurrentemente, no se quiere que un

proceso se interfiera con otro, o con el sistema operativo mismo. La protección involucra que todos los accesos a los recursos son controlados. La seguridad del sistema con el exterior es también importante. Tal seguridad comienza con que cada usuario tenga su password, para que se le permita entrar al sistema y así poder utilizar los recursos. Esto también se extiende en la defensa de dispositivos de I/O externos, incluyendo modems y adaptadores de red, del intento de accesos inválidos.

System Calls

Las llamadas al sistema proveen la interface entre un proceso y el sistema operativo. Estas llamadas están generalmente disponibles como instrucciones en lenguaje assembler. Algunos sistemas permiten realizar llamadas al sistema en lenguaje de alto nivel, en donde luego se transforman en llamadas de bajo nivel. Igualmente, lenguajes como C, PL, etc, permiten llamadas al sistema haciéndolas directamente. Por ejemplo examinemos la función READ. Esta llamada tiene tres parámetros: el primero especifica el archivo, el segundo especifica el buffer, y el tercero especifica el número de bytes por leer. Una llamada de READ desde un programa en C podría verse como:

```
cuenta = read(file, buffer, nbytes);
```

La llamada al sistema (y el procedimiento de biblioteca) devuelve en cuenta el número de bytes que realmente se leyeron. Este número normalmente es igual a nbytes pero es menor en caso de que se haya llegado al final del archivo durante la lectura. Si la llamada al sistema no se ejecutó, ya sea por un mal parámetro o por problema de disco, entonces cuenta tendrá el valor –1 y una variable global llamada *errno* tendrá el número de error.

Como ejemplo de cómo son usadas las llamadas al sistema, consideremos la escritura de un programa simple el cual lee datos desde un archivo y copia los mismos en otro archivo. La primer entrada que el programa necesitara es los nombre de los dos archivos. La obtención de estos nombres se puede lograr de diferentes maneras, dependiendo del diseño del sistema operativo. Una forma es que el programa pregunte al usuario por los nombres de los archivos. En un sistema interactivo, el acceso de los nombres requerirá una secuencia de llamadas al sistema, primero para escribir un mensaje en la pantalla, y luego para leer del teclado los dos nombres de los archivos. Otro tipo de acceso, utilizado por los sistemas batch, es el de especificar los nombres de los archivos con sentencias de control. En este caso, se debe tener un mecanismo de pasaje de parámetros desde la sentencia de control al programa que los necesita. En un sistema basado en mouse e iconos, un menú con los nombres de los archivos puede ser mostrado en una ventana. El usuario puede entonces utilizar el mouse para seleccionar el nombre de los archivos.

Una vez que se tiene los nombres de los archivos, el programa debe abrir el archivo de entrada y crear el de salida. Cada una de estas operaciones requiere una llamada al sistema. Hay también condiciones de errores posibles para cada operación. Cuando el programa trata de abrir el archivo de entrada, éste puede encontrar que no existe un archivo con tal nombre o que el archivo esta protegido contra cualquier tipo de acceso. En estos casos, el programa imprimirá un mensaje en el monitor (otra secuencia de llamadas al sistema) y terminara anormalmente (otra llamada al sistema). En caso de que el archivo exista, entonces se debe crear el archivo de salida. Se puede encontrar en este punto que ya existe un archivo con este nombre, con lo cual puede causar que el programa aborte (una llamada al sistema), o se puede eliminar el archivo existente (otra llamada al sistema). Otra opción, en un sistema interactivo, es la de preguntar al usuario (una secuencia de llamadas al sistema para presentar el mensaje de pregunta y para leer la respuesta del usuario) entre reemplazar el archivo existente o el de terminar el programa.

Ahora que ya se tiene los dos archivos disponibles, se entra en un loop el cual lee del archivo de entrada (una llamada al sistema) y escribe en el archivo de salida (otra llamada al sistema). Cada lectura y escritura debe retornar la información de estado para resguardarse de posibles condiciones de error. En una entrada, el programa puede encontrar el final del archivo, o que se produzca un fallo de hardware en la lectura (tal como un error de paridad). La operación de escritura también puede encontrar varios errores, dependiendo del dispositivo de salida (no existe más espacio en el disco, la impresora no tiene papel, y así).

Finalmente, luego de que se copio el archivo, el programa debe cerrar ambos archivos (otra llamada al sistema), escribir un mensaje en la consola (más llamadas al sistema), y finalizar normalmente (la ultima llamada). La mayoría de los usuarios no ven este nivel de detalle.
Tres métodos generales se utilizan para el pasaje de parámetros al sistema operativo. La más simple es la de pasar los parámetros en registros. En algunos casos, sin embargo, puede haber más parámetros que registros. En estos casos, los parámetros son almacenados en un bloque o tabla en la memoria, y la dirección del bloque es pasada como parámetro en un registro. Los parámetros también pueden ser ubicados en una pila por el programa, y sacados de la pila por el sistema operativo. Algunos sistemas operativos prefieren los sistemas de bloque o pila ya que con ellos no existe limite de la cantidad de parámetros que se vayan a pasar.
Las llamadas al sistema pueden ser agrupadas en 5 categorías.

Control de procesos	finalizar, abortar
	cargar, ejecutar
	crear, terminar procesos
	dar atributos de procesos, setear atributos de los procesos
	esperar por tiempo
	esperar por evento, señal de evento
	asignar y liberar memoria
Manipulación de archivos	crear, eliminar archivos
	abrir, cerrar
	leer, escribir. reponer
	dar atributos de archivos, setear atributos de los archivos
Manipulación de dispositivos	pedido por un dispositivo, liberar el dispositivo
	leer, escribir. reponer
	dar atributos de dispositivos, setear atributos de los dispositivos
	unir o separar dispositivo
Mantenimiento de la información	dar día u hora, setear el día o la hora
	dar datos del sistema, setear datos del sistema
	dar atributos de procesos, archivos o dispositivos
	setear atributos de procesos, archivos o dispositivos
Comunicaciones	crear u eliminar la conexión de una comunicación
	enviar, recibir mensajes
	transferir estado de la información
	unir o separar dispositivos remotos

Control de trabajos y de procesos: un programa en ejecución necesita parar su ejecución ya sea de forma normal (*end*) o de forma anormal (*abort*).
Ya sea que el programa termine de forma normal o anormal, el sistema operativo debe transferir el control al interprete de comandos. El interprete de comandos lee entonces el siguiente comando. En un sistema interactivo, el interprete de comandos simplemente continua con el siguiente comando, este asume que el usuario emitirá un comando apropiado para responder a cualquier error. En un sistema de lotes, el interprete de comandos usualmente termina el trabajo entero y continua con el siguiente.
Un proceso o trabajo ejecutando un programa puede desear cargar (**load**) o ejecutar (**execute**) otro programa. Esta característica permite al interprete de comandos ejecutar un programa directamente por, por ejemplo, un comando del usuario, el click del mouse, o un comando de un lote. Una cuestión interesante es donde se retorna el control cuando el programa cargado finaliza. Esta cuestión esta relacionada al problema de si el programa existente es perdido, almacenado, o permitir que continúe la ejecución concurrentemente con el nuevo programa.
Si el control retorna al programa existente cuando el nuevo programa termina, se debe almacenar la imagen de la memoria del programa existente; así, hemos creado un mecanismo para que un programa llame a otro. En caso de que ambos programas continúen concurrentemente, hemos creado un nuevo proceso o trabajo para ser multiprogramado. A menudo, existe una llamada al sistema para este propósito (**create process** o **submit job**).
En caso de que creemos un nuevo trabajo o proceso, o quizá aun un conjunto de trabajos y procesos, debemos ser capaces de controlar su ejecución. Este control requiere la habilidad de determinar y resetear los atributos de un trabajo o proceso, incluyendo la prioridad del trabajo, su máximo tiempo de

ejecución disponible, y otros (**get procces attributes** y **set process attributes**). También puede que deseamos terminar un trabajo o proceso que creamos (**terminate process**) en caso de encontrar que era incorrecto, o ya no se necesita.
Luego de crear los procesos o trabajos, puede que necesitemos esperar que ellos finalicen su ejecución. Podemos desear esperar por una determinada cantidad de tiempo (**wait time**), o deseamos esperar hasta que un determinado evento ocurra (**wait event**). Los trabajos o procesos deben señalar cuando el evento ha ocurrido (**signal event**). Llamadas al sistema de este tipo que tratan con la coordinación de procesos concurrentes se discuten en el capítulo 6.
Existen muchas variaciones en el control de trabajos y procesos. Veamos algunos ejemplos. El sistema operativo MS-DOS es un ejemplo de un sistema de única tarea, el cual tiene un interprete de comando que se invoca en el momento que se inicia la computadora (Figura 3.3a). Ya que MS-DOS es un sistema de solo una tarea, éste usa un método simple para correr un programa, y no crea un proceso nuevo. Este carga el programa en la memoria y otorga al programa tanta memoria como este disponible. Puede que hasta sobreescriba la parte designada por el interprete de comandos (Figura 3.3b). Luego se setea el puntero de instrucción a la primer instrucción del programa. Se corre entonces el programa el cual puede que un error cause un trap, o que ejecute un system call para finalizar. En ambos casos, el código que produjo el error es almacenado en el sistema de memoria para su uso posterior. Siguiendo con la acción, la pequeña porción del interprete que no fue sobre-escrita continúa la ejecución: su primera tarea es recargar el resto del interprete de comandos desde el disco. Luego de que se completa esta tarea, el interprete de comandos presenta al usuario el código que produjo el error previo, o al siguiente programa (hay casos que se presenta el error al programa que le sigue para así este toma diferentes acciones).

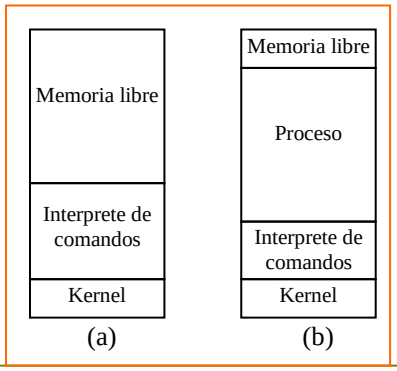


Figura 3.3 Ejecución en MS-DOS. (a) Cuando el sistema inicia. (b) Ejecutando un programa.

Aunque el MS-DOS no tiene multitarea, provee un método para limitar la ejecución concurrente. Un programa TSR es un programa que "caza" una interrupción, y sale luego con la llamada al sistema ***terminate and stay resident***. Por ejemplo, éste programa puede cazar la interrupción del reloj por medio de la ubicación de la dirección de una de sus subrutinas en la lista de rutinas de interrupciones para ser llamada cuando el timer del sistema lo desencadene. De esta forma, la rutina TSR será ejecutada varias veces por segundo, en cada tic del reloj. La llamada al sistema ***terminate and stay resident*** causa que el MS-DOS reserve espacio para ser ocupado por el TSR, por lo que éste no será sobreescribido cuando el interprete de comando sea recargado.
UNIX sí es un ejemplo de un sistema multitarea. Cuando un usuario se ingresa al sistema (log on), el shell (interprete de comando) del usuario es corrido. Este shell es similar al shell del MS-DOS en que éste acepta comandos y ejecuta programas que el usuario le pide. Sin embargo, ya que UNIX es un sistema multitarea, el interprete de comando puede continuar corriendo mientras otra tarea esta siendo corrida (Figura 3.4).

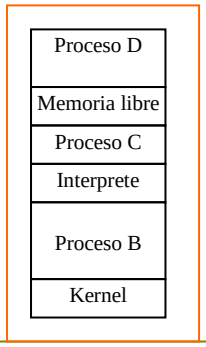


Figura 3.4 UNIX ejecutando múltiples programas.

Para comenzar un nuevo proceso (es decir, para crearlo), el shell ejecuta una llamada al sistema ***fork***. Luego, el programa seleccionado es cargado en la memoria vía una llamada al sistema ***exec***, y el programa es entonces ejecutado. Dependiendo de la forma en que el comando fue pedido, el shell puede esperar que el proceso termine, o correr el proceso en segundo plano. En el último caso, el shell inmediatamente pide otro comando. Cuando un proceso es corrido en segundo plano, éste no puede recibir entradas directamente desde el teclado, ya que el shell está usando dicho recurso. La I/O es de esta forma hecha a través de archivos, o por el mouse en un sistema de ventanas. Mientras tanto, el usuario esta libre de pedir al shell que corra otro programa, que se le muestre el progreso de los procesos corriendo, cambiar la prioridad de programas, y demás. Cuando el proceso se completa, éste ejecuta una llamada al sistema ***exit*** para terminar, retornando al proceso invocador (es decir, al que emitió el fork) un 0 en caso de que no hubiera error, o un valor distinto de 0 en caso de error. Este estado, o código de error se deja disponible al shell u a otros programas.

Manipulación de archivos: En un sistema de archivos lo primero que se debe poder hacer es crear y eliminar archivos (***create*** y ***delete***). Ambas llamadas al sistema requieren el nombre del archivo y quizá algún atributo del mismo. Una vez que se creo, se necesita abrirlo (***open***) para ser usado. Podemos también leer (***read***), escribir (***write***), o reposicionarnos (***reposition***) (retroceder o avanzar más de una posición en el archivo, por ejemplo). Finalmente, se necesita cerrar (***close***) el archivo, indicando que éste no será más usado.
Se necesitan también éstas mismas operaciones para el manejo de directorios en caso de que nuestra estructura para organizar los archivos en el sistema sea de éste tipo. Además, ya sea para archivos o para directorios, debemos ser capaces de determinar el valor de varios atributos, o quizá deseamos cambiar alguno de éstos atributos. Los atributos de los archivos pueden ser su nombre, el tipo, los códigos de protección, cantidad de información, etc. Las ultimas dos llamadas al sistema, ***get file attribute*** y ***set file attribute*** logran esta función.

Administración de dispositivos: Un programa, cuando éste está corriendo, puede necesitar algunos recursos adicionales para proceder su ejecución. Algunos recursos adicionales pueden ser más memoria, acceso a archivos, etc. En caso de que estos recursos estén disponibles, pueden ser concedidos, y el control se retorna al programa; de otra forma, el programa deberá esperar hasta que los recursos que necesita estén disponibles.
Los archivos se pueden ver como dispositivos abstractos o virtuales. Así, muchas de las llamadas al sistema de archivos se necesitan también para dispositivos. Si hay muchos usuarios en el sistema, sin embargo, se debe primero pedir (***request***) el dispositivo, asegurar el uso exclusivo de éste. Luego de que se finalizo con su uso, se debe liberar (***release***). Estas funciones son similares a las llamadas al sistema ***open*** y ***close***.
Luego de que el dispositivo a sido pedido (y asignado para nosotros), podemos leer (read), escribir (write), y posiblemente reubicarnos (reposition), así como lo hacemos en archivos ordinarios. De hecho, las similitudes entre dispositivos de I/O y los archivos son tan grandes que muchos sistemas operativos

(incluyendo MS-DOS y UNIX) unen los dos en una combinación de estructura archivo-dispositivo. En este caso, los dispositivos de I/O son identificados por nombre de archivos especiales.

Sustento de la información: Muchas llamadas al sistema existen simplemente para la transferencia de la información entre programas de usuarios y el sistema operativo. Por ejemplo, la mayoría de los sistemas tiene una llamada al sistema que retorna la hora y el día (***time*** y ***date***). Otras llamadas al sistema retornan información sobre el sistema, tal como el número de usuarios actuales, el número de versión del sistema operativo, la cantidad de memoria libre o espacio de disco, etc.
Además, el sistema operativo mantiene información sobre todos éstos procesos, y hay llamadas al sistema para acceder a esta información. Generalmente, hay también llamadas para resetear la información del proceso (***get process attibutes*** y ***set process attributes***).

Comunicación: Existen dos modelos comunes de comunicación. En el modelo de pasaje de mensajes, la información es intercambiada a través de la comunicación entre procesos proveída por el sistema operativo. Antes de que tome lugar la comunicación, se debe abrir la conexión. Se debe conocer el nombre del otro comunicador, el cual puede estar en la misma CPU, o puede ser un proceso en otra computadora conectado por una comunicación vía red. Cada computadora en una red tiene su nombre de host el cual es por éste nombre que se conoce a un computadora. Además, cada proceso en una computadora tiene su propio nombre, el cual es con ambos datos que un sistema operativo se puede referir a ellos. Las llamadas al sistema ***get hostid***, y ***get processid*** hacen esta translación. Estos identificadores son pasados a las llamadas ***open*** y ***close***, proveídas por el sistema de archivos, o para las llamadas especificas ***open connection*** y ***close connection***, dependiendo del modelo de comunicación del sistema. El proceso recibidor debe dar su aceptación de la comunicación vía una llamada al sistema ***accept connection***. La mayoría de los procesos ejecutan una llamada ***wait for connection*** y son despertados cuando se hace una conexión. La fuente de la conexión es conocida como el cliente, y el que recibe los mensajes es conocida como servidor, e intercambian mensajes vía llamadas al sistema ***read message*** y ***write message***. La comunicación es terminada por la llamada ***close connection***.
En los modelos de memoria compartida, los procesos utilizan llamadas al sistema ***map memory*** para ganar el acceso a los espacios de memoria propios de otros procesos. Se debe tener en claro que el sistema operativo siempre trata de prevenir que un proceso ingrese en el espacio de memoria de otro. En el modelo de memoria compartida, ambos procesos deben estar de acuerdo en eliminar esta restricción. Así se puede lograr que se intercambie información por medio de la lectura y escritura de éstas partes de memoria compartida.
Algunos sistemas operativos implementan ambos modelos. El traspaso de mensajes es muy útil cuando se necesita intercambiar pequeñas cantidades de datos. El modelo de memoria compartida permite una máxima velocidad, y se puede realizar a la velocidad de la memoria cuando se produce en la misma computadora.

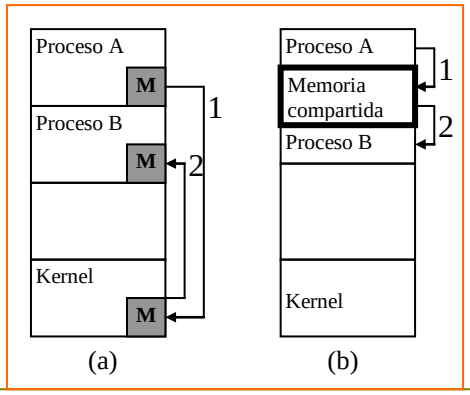


Figura 3.5 Modelos de comunicación. (a) Traspaso de mensajes. (b) Memoria compartida.

Programas del sistema

Otro aspecto de los sistemas operativos modernos es la colección de programas del sistema. Como se ve en la figura 1.1, los programas del sistema se encuentran sobre el sistema operativo y favorecen el ambiente en que se desarrollan los programas y se ejecutan. Estos programas se pueden dividir en varios grupos:

- *Manipulación de archivos:* Estos programas crean, eliminan, copian, renombran, imprimen y listan archivos y directorios.
- *Información de estado:* Algunos programas simplemente preguntan al sistema por la hora, el día, la cantidad de memoria disponible o el espacio de disco, el número de usuarios, etc.
- *Modificación de archivos:* Varios editores de textos pueden estar disponibles para crear y modificar los contenidos de los archivos almacenados en disco o cinta.
- *Soportar lenguajes de programación:* Compiladores, assemblers, y interpretes de lenguajes de programación comunes (PASCAL, BASIC, C y LISP) están disponibles a los usuarios con el sistema operativo. Muchos de estos programas ahora están proveídos separadamente.
- *Carga y ejecución de programas:* Luego de que el programa es compilado, debe ser cargado en la memoria para la ejecución. El sistema puede proveer cargadores, reubicadores, linkeadores, etc.
- *Comunicaciones:* Estos programas proveen el mecanismo para la creación de conexiones virtuales entre procesos, usuarios, y diferentes sistemas de computadoras. Estos programas permiten a los usuarios enviar mensajes (como mail's), o la transferencia de archivos de una máquina a otra.

La mayoría de los sistemas operativos están cargados con programas que son muy útiles para resolver este tipo de problemas. Tales programas incluyen web browsers, formateadores de texto, sistemas de base de datos, compiladores, juegos, etc. Estos programas son conocidos como programas de aplicación. Quizá el programa más importante del sistema para el sistema operativo es el intérprete de comandos.

Muchos comandos manipulan archivos: crean, eliminan, listan, imprimen, copian, ejecutan, etc. Existen dos formas generales en el cual éstos programas pueden ser implementados. En una, el interprete de comandos contiene el código para ejecutar el comando. Por ejemplo, un comando para eliminar un archivo puede causar que el interprete de comandos salte a la sección de su código que establece los parámetros y hace la llamada al sistema correspondiente. En este caso, el número de comandos que son dados determinan el tamaño del interprete de comandos.

Otra alternativa usada por UNIX, además de otros sistemas operativos, implementa la mayoría de los comandos por medio de programas del sistema especiales. En este caso, el intérprete de comando no conoce de ninguna manera el comando, éste simplemente usa el comando para identificar el nombre del archivo que debe ser cargado en la memoria y ejecutado. Así, el comando:

delete *G*

provocaría que se busque un archivo llamado **delete**, que se cargue dicho archivo en la memoria y sea ejecutado con el parámetro *G*. De ésta forma, los programadores pueden agregar nuevos comandos al sistema fácilmente creando nuevos archivos con los nombres apropiados. El programa interprete de comandos, el cual puede ser ahora bastante pequeño, no debe ser cambiado en caso de que se agreguen nuevos comandos.

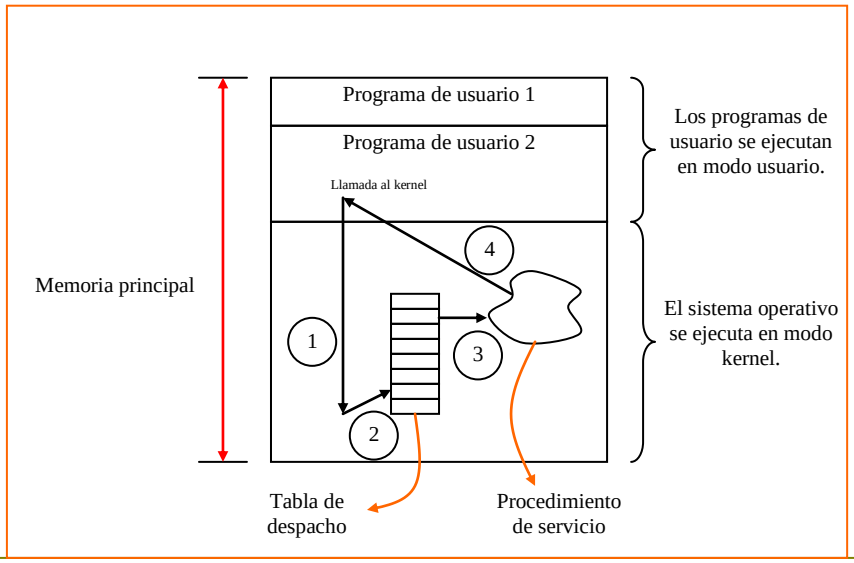
Hay problemas en el momento de diseñar un interprete de comandos con esta última idea. Notemos primero que, ya que el código para ejecutar un comando es un programa de sistema separado, el sistema operativo debe proveer un mecanismo para el pasaje de parámetros desde el interprete de comandos al programa del sistema. Esta tarea puede ser a menudo difícil de manejar, ya que el interprete de comandos y el programa del sistema pueden no estar ambos en memoria al mismo tiempo, y la lista de parámetros puede ser larga. Además, es más lento cargar y ejecutar un programa que simplemente saltar a otra sección del código en el mismo programa.

A continuación en todo lo que resta del apunte no se distinguirá entre programas del usuario y programas del sistema.

Estructura del sistema

Un sistema tan grande y complejo como un sistema operativo debe cumplir su función adecuadamente y debe ser fácil de modificar. Una idea básica es la de dividir la tarea en pequeños componentes en lugar de tener un sistema monolítico. Cada uno de estos módulos debe ser una porción bien definida del sistema, con sus entradas, salidas y funciones bien definidas. Ya se vieron los componentes comunes de todo sistema operativo, ahora discutiremos la forma en que estos componentes son interconectados y ubicados en el kernel.

Estructura simple: En estos sistemas de estructura simple (llamados sistemas monolíticos), el sistema operativo se escribe como una colección de procedimientos, cada uno de los cuales puede invocar a cualquiera de los otros cuando necesite hacerlo. Cuando se usa esta técnica, cada procedimiento del sistema tiene una interfaz bien definida en términos de parámetros y resultados, y cada uno está en libertad de invocar a cualquier otro. Para construir el programa objeto real del sistema operativo cuando se utiliza este enfoque, lo primero que se hace es compilar todos los procedimientos individuales, o archivos que contienen los procedimientos, y luego se vincula en un solo archivo objeto usando el linker del sistema. Prácticamente no existe la ocultación de la información, todos los procedimientos son visibles para todos los demás (en contraposición a una estructura que contiene módulos o paquetes, en la que gran parte de la información se oculta dentro de los módulos y, solo se puede invocar desde fuera del módulo los puntos de entrada designados oficialmente).
No obstante, en estos sistemas se tiene un poco de estructura. Las llamadas al sistema proporcionadas por el sistema operativo se solicitan colocando los parámetros en lugares bien definidos y ejecutando luego un trap. Este trap cambia la máquina de modo usuario a modo kernel y transfiere el control al sistema operativo, mostrándose con el evento número (1) en la figura. Luego el sistema operativo examina los parámetros de la llamada para determinar cual llamada al sistema se ejecutará; esto se muestra en el punto (2). Luego el sistema operativo consulta una tabla que en la entrada número *K* un puntero al procedimiento que lleva a cabo la llamada al sistema *K*. Ésta operación, marcada con el punto (3), identifica el procedimiento de servicio y lo invoca. Cuando se completa el trabajo y se termina la llamada al sistema, el control se retorna al programa del usuario (punto (4)) para que continúe su ejecución en la instrucción siguiente a la llamada al sistema.



Pasos para realizarse una llamada al sistema: (1) El programa de usuario entra en el kernel por medio de un trap. (2) El sistema operativo determina el número de servicio requerido. (3) El sistema operativo invoca el procedimiento de servicio. (4) Se retorna el control al programa de usuario.

Existen varios sistemas comerciales que no tiene una estructura bien definida. Un ejemplo de estos sistemas es el MS-DOS, el cual fue originalmente diseñado por un conjunto de personas que no tenían ni idea de lo famoso y útil que se transformaría su sistema. En la figura 3.6 se ve su estructura general.

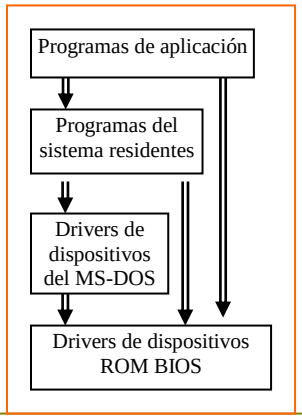


Figura 3.6 Estructura de capas de MS-DOS.

En MS-DOS, las interfases y la funcionalidad de cada nivel no están bien definidos. Por ejemplo, los programas de aplicación son capaces de acceder a las rutinas básicas de I/O para escribir directamente en los drivers del monitor y de los discos. Tales libertades dejan lugar al sistema operativo a errores de programas, causando que el sistema completo falle cuando un programa falla. Por supuesto, MS-DOS fue limitado por el hardware de su era, por lo que los diseñadores no tuvieron elección en el momento de dejar tales libertades.

Otro ejemplo es el sistema operativo original de UNIX, el cual también fue limitado por el hardware de su época. Este sistema consiste de dos partes separadas: el kernel y los programas del sistema. El kernel es dividido además en una serie de interfaces y drivers de dispositivos, los cuales se han ido agregando a través de los años. Una vista aproximada de este sistema se ve en la figura 3.7. Todo lo que esta debajo de la interfase de las llamadas al sistema y sobre el hardware físico es el kernel. El kernel provee el sistema de archivos, scheduling de CPU, administración de la memoria, y otras funciones del sistema operativo por medio de llamadas al sistema. Todo junto compone una inmensa cantidad de funcionalidad que es combinada en un solo nivel. Los programas del sistema usan las llamadas al sistema soportadas por el kernel para proveer funciones útiles, tales como la compilación y la manipulación de archivos. Las llamadas al sistema definen la interfaz del programador a UNIX; el conjunto de programas del sistema comúnmente disponibles definen la interfaz del usuario. Las interfaces del usuario y el programador definen el contexto que el kernel debe soportar. Varias versiones de UNIX se han desarrollado en el cual el kernel se particiona además entre limitaciones funcionales. El sistema operativo AIX (versión IBM de UNIX) separaba el kernel en dos partes. Mach reduce el kernel en un pequeño conjunto de funciones principales moviendo todas las funciones no esenciales a programas del sistema y aún a programas a nivel de usuario. El resto es un sistema operativo *microkernel* implementando solo un pequeño conjunto de primitivas necesarias.

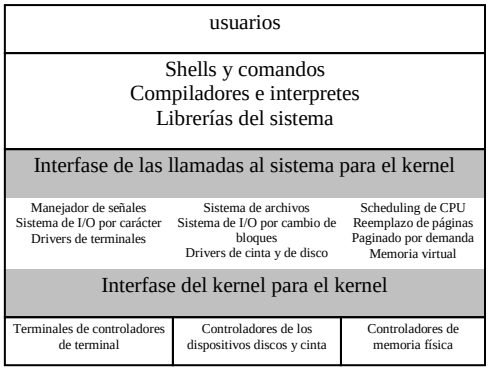


Figura 3.7 Estructura del sistema UNIX.

Estilo de capas: Las nuevas versiones de UNIX están diseñadas para usar un hardware más avanzado. Con el soporte del hardware, los sistemas operativos se pueden dividir en pequeñas y más apropiadas piezas que aquellas de los antiguos sistemas UNIX O MS-DOS. Así, el sistema operativo puede tener un mayor

control de la computadora y de las aplicaciones que hacen uso de la computadora. Los diseñadores tienen mayor libertad en hacer cambios en la parte interna del sistema. Técnicas familiares se usan para ayudar en la creación de sistemas operativos modulares. Bajo la idea top-down, la funcionalidad global y las características se pueden determinar y separar en componentes.
La modularización del sistema se puede lograr de varias formas: la más atrayente es la de capas, el cual consiste de romper el sistema operativo en un número de capas (niveles), cada una construida encima de niveles inferiores. La capa de más abajo (capa 0) es el hardware, mientras que la más alta (capa N) es la interfase del usuario.
Un sistema operativo de capas es una implementación de un objeto abstracto el cual es la encapsulación de datos y operaciones que manipulan dichos datos. La mayor ventaja del estilo de capas es la modularidad. Las capas son seleccionadas de manera que cada una use las funciones (operaciones) y servicios sólo de las capas del nivel inferior. Este estilo simplifica el debugged (corrección de errores) y la verificación del sistema. La primer capa puede ser debugged independientemente del resto del sistema ya que, por definición, esta capa solo usa el hardware (el cual se asume que es correcto) para implementar sus funciones. Una vez que la primer capa es debugged, se puede asumir que su funcionalidad es correcta mientras la segunda capa esta trabajando, y así con las demás. Si se encuentra un error en alguna capa mientras está siendo debugged, entonces de seguro el error estará en dicha capa ya que las anteriores ya fueron debugged.
Cada capa se implementa usando solo las operaciones que proveen las capas que están en el nivel inmediatamente inferior. Una capa no necesita conocer como se implementan éstas funciones, sino que solo necesita saber que es lo que la función hace. De hecho, cada capa esconde la existencia de ciertas estructuras de datos, operaciones y hardware a las capas de niveles superiores.

Maquinas virtuales

El sistema de una computadora esta constituido de capas. El hardware es el nivel más bajo de tales sistemas. El kernel, corriendo en el siguiente nivel, usa las instrucciones de hardware para crear un conjunto de llamadas al sistema para ser usadas por las capas externas. Los programas del sistema arriba del kernel son, por lo tanto, capaces de usar ya sea las llamadas al sistema o las instrucciones del hardware. Así, los programas del sistema tratan al hardware y a las llamadas al sistema como que si ambos estén en el mismo nivel.
Algunos sistemas llevan este esquema un nivel más allá por medio de que los programas del sistema puedan ser llamados fácilmente por los programas de aplicación. Como antes, aunque los programas del sistema están en un nivel más alto que las otras rutinas, los programas de aplicación pueden ver cualquier cosa bajo de ellos en la jerarquía. Este es el concepto de maquina virtual.
Por medio de la utilización de un scheduling de CPU, y técnicas de memoria virtual, un sistema operativo puede crear la ilusión de múltiples procesos, cada uno corriendo en su propio procesador con su propia memoria (virtual). En una maquina virtual, cada proceso esta proveído con una copia (virtual) de la parte más importante de la computadora (Figura 3.12).

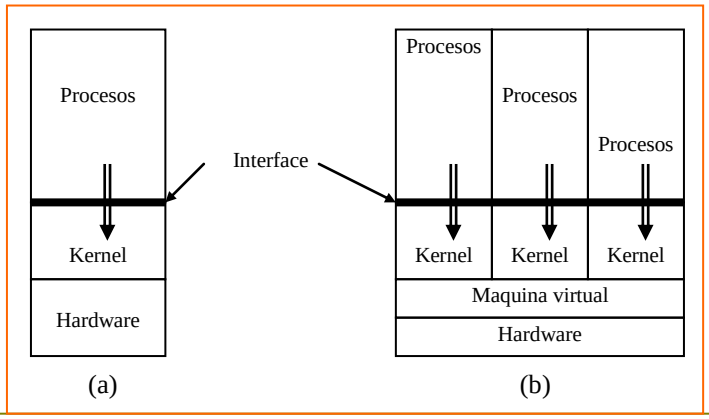


Figura 3.12 Modelos de sistema. (a) Maquina no virtual. (b) Maquina virtual.

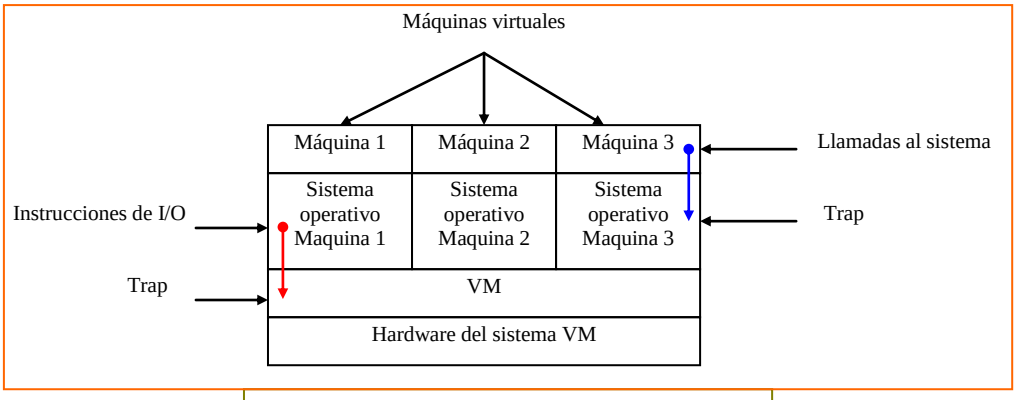
Los recursos físicos de la computadora son compartidos para crear las maquinas virtuales. El scheduling de la CPU puede ser usado para compartir la CPU y para crear la apariencia de que los usuarios tienen su propio procesador. El spooling y un sistema de archivos proveen lectores virtuales y líneas de impresoras virtuales.

La mayor dificultad con este método de maquinas virtuales involucra a los sistemas de disco. Supongamos que la maquina física tiene tres drivers de disco pero quiere soportar siete maquinas virtuales. Claramente, no se puede asignar un drive de disco para cada maquina virtual. Recordemos que el software de la maquina virtual por sí mismo necesitara espacio de disco para proveer memoria virtual y spooling. La solución es proveer discos virtuales, los cuales son todos idénticos en todo excepto en el tamaño. Existen minidiscos en el sistema operativo VM de IBM. El sistema implementa cada minidisco asignando tantas pistas como el minidisco necesite en los discos físicos. Obviamente, la suma de los tamaños de todos los minidiscos debe ser menor que la actual cantidad de espacio de disco físico disponible.

De este modo, se les da a los usuarios su propia maquina virtual. Ellos pueden correr entonces cualquiera de los sistemas operativos o paquetes de software que están disponibles en la maquina fundamental (es decir, todos los paquetes que estén en el disco de la maquina en su conjunto).

Analicemos un poco el sistema Vm de IBM. Este sistema se basa en una observación: un sistema de tiempo compartido ofrece (1) multiprogramación y (2) una máquina extendida con una interfaz más cómoda que el hardware mismo. La esencia de VM consiste en separar por completo éstas dos funciones.

El corazón del sistema, conocido como **monitor de máquina kernel**, se ejecuta en el hardware mismo y realiza la multiprogramación, proporcionando no una, sino varias máquina virtuales a la siguiente capa superior (ver figura).



Estructura del sistema VM

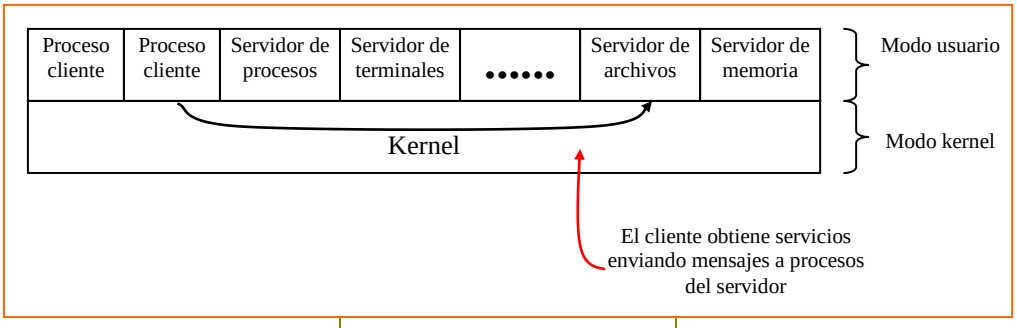
Cada máquina virtual puede ejecutar cualquier sistema operativo. Diferente máquinas virtuales pueden ejecutar diferentes sistemas operativos. Cuando un programa de usuario ejecuta una llamada al sistema, la llamada produce un trap al sistema operativo de su máquina virtual, no al sistema VM, tal como se haría si el programa se estuviera ejecutando en una máquina real en lugar de en una máquina virtual. A

continuación, el sistema operativo de la máquina virtual emite las instrucciones de I/O del hardware para leer su disco virtual, o lo que sea que se necesite para llevar a cabo la llamada al sistema. Estas instrucciones de I/O producen un trap al sistema VM, el cual, entonces, las ejecuta como parte de su simulación de hardware real. Al separar por completo las funciones de multiprogramación y su suministro de máquina extendida, cada uno de los componentes puede ser mucho más sencillo, flexible, y fácil de mantener.

Modelo cliente-servidor

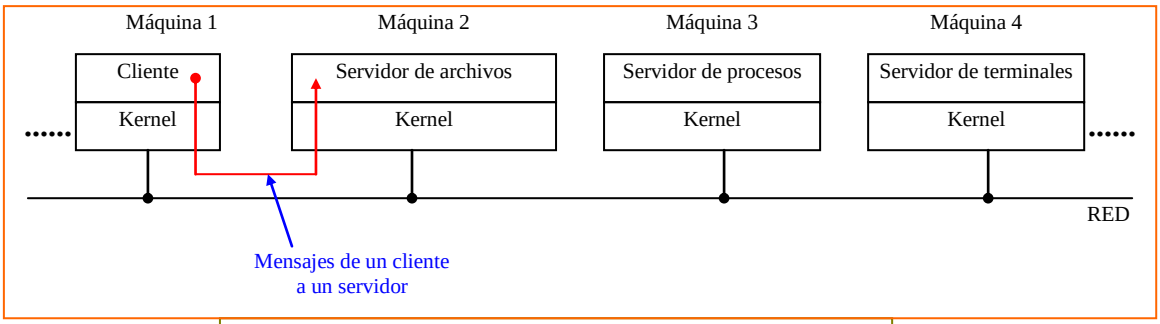
Una tendencia en los sistemas operativos modernos es llevar aún más lejos ésta idea de trasladar código a capas superiores y quitarle lo más que se pueda al sistema operativo, dejando un kernel mínimo. Un enfoque usual consiste en implementar la mayor parte de las funciones del sistema operativo en procesos de usuarios. Por ejemplo, para solicitar un servicio de leer un bloque de un archivo, un proceso de usuario (ahora llamado **proceso cliente**) envía la solicitud a un **proceso servidor**, el cual realiza el trabajo y retorna la respuesta.

En este modelo, que se muestra en la figura de abajo, lo único que el kernel hace es manejar la comunicación entre los clientes y los servidores. Al dividir el sistema operativo en partes, cada una de las cuales se encarga de una función del sistema, como el servidor de archivos, de procesos, de terminales o de memoria, cada parte puede ser pequeña y manejable. Además, dado que todos los servidores se ejecutan como procesos en modo usuario, y no en modo kernel, no tienen acceso directo al hardware. Por lo tanto, si se produce un error en el servidor de archivos, es posible que el mismo se caiga, pero no provocará que se caiga todo el sistema.



Modelo cliente-servidor

Otra ventaja de este modelo cliente-servidor es su adaptabilidad para usarse en sistemas distribuidos (ver figura de abajo). Si un cliente se comunica con el servidor enviándole mensajes, el cliente no necesita saber si el mensaje será atendido localmente en su propia máquina o si se envió a través de la red a un servidor en una máquina remota. En lo que al cliente respecta, sucede lo mismo en ambos casos: se envía la solicitud y se obtiene una respuesta



Modelo cliente-servidor en un sistema distribuido.

II. Administración de procesos

Un proceso es un programa en ejecución, y necesitara ciertos recursos (tal como tiempo de CPU, memoria, archivos, dispositivos de I/O, etc) para llevar a cabo su tarea. Estos recursos son asignados al proceso o cuando éste es creado o mientras está siendo ejecutado.
Un sistema consiste de una colección de procesos. Tal colección esta formada por los procesos del sistema operativo, el cual ejecutan código del sistema; y procesos del usuario, el cual ejecutan código del usuario. Todos estos procesos pueden ejecutar concurrentemente.
En los sistemas de tiempo compartido, periódicamente el sistema operativo decide dejar de ejecutar un proceso y comenzar a ejecutar otro, por ejemplo, porque al primero se le termino el tiempo de CPU. Ante esto, y para que más tarde el proceso pueda continuar su ejecución, se debe almacenar información del proceso, el cual puede consistir de un espacio de direcciones, llamado imagen de núcleo, y registros.
El sistema operativo es responsable de las siguientes actividades con respecto a la administración de procesos: la creación y eliminación tanto de procesos del sistema como del usuario, el scheduling de procesos, y la provisión de mecanismos de sincronización, comunicación, y el manejo de deadlocks para procesos.

4. Procesos

En un principio, las computadoras permitían que solo un programa sea ejecutado a la vez. Este programa tenia completo control del sistema y tenia acceso a todos los recursos del sistema. Actualmente, los sistemas modernos permiten que muchos procesos puedan ser cargados en la memoria y ejecutados concurrentemente.

Concepto de proceso

En el desarrollo de este capítulo se usará indistintamente la palabra trabajo y proceso.

El proceso: Informalmente, un proceso es un programa en ejecución. La ejecución de un proceso debe ser en forma secuencial, esto es, en cualquier momento como mucho una instrucción es ejecutada.
Un proceso es más que el código del programa (a veces conocido como la sección de texto). Esto también incluye la actividad actual, representada por el valor del contador del programa y los contenidos de los registros del procesador. Un proceso también incluye la pila del proceso, conteniendo los datos temporarios (tal como los parámetros de las subrutinas, direcciones de retorno, y variables temporales), y una sección de datos conteniendo variables globales.
Se debe tener en claro que un programa por sí solo no es un proceso; un programa es una entidad pasiva, tal como el contenido de los archivos almacenados en disco, mientras que un proceso es una entidad activa, con un contador de programa conteniendo la siguiente instrucción a ejecutar y un conjunto de recursos asociados.
Aunque dos procesos pueden ser asociados con un mismo programa, ellos son sin embargo considerados dos secuencias separadas de ejecución. Por ejemplo, varios usuarios pueden estar corriendo copias del programa de mail, o el mismo usuario puede invocar muchas copias de un editor. Cada uno de estos es un programa separado y, aunque las secciones de texto son equivalentes, la sección de datos variará. Esto también es común cuando un proceso produce muchos procesos cuando se está ejecutando.

Estado del Proceso: Cuando un proceso esta en ejecución, éste cambia de estado. El estado de un proceso es definido por la actividad actual que esta haciendo. Cada proceso puede estar en un de los siguientes estados:

- *Nuevo:* el proceso esta siendo creado.
- *Corriendo:* las instrucciones están siendo ejecutadas.

- *Esperando:* el proceso esta esperando que ocurra un evento (tal como la completitud de una I/O, o la recepción de una señal).
- *Listo:* el proceso esta esperando que se le asigne el procesador.
- *Terminado:* el proceso ha finalizado su ejecución.

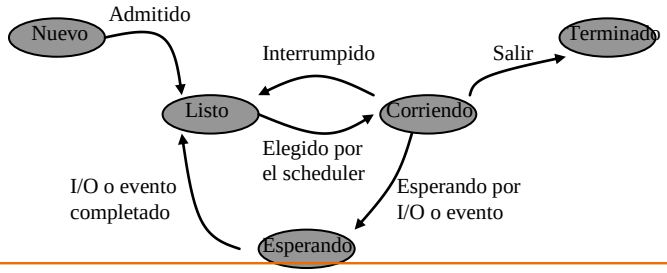


Figura 4.1 Diagrama del estado de un proceso.

Bloque de control del proceso: Cada proceso es representado en el sistema operativo por un bloque de control de proceso (PCB: Process Control Block) (Figura 4.2).

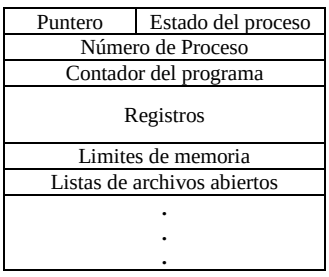


Figura 4.2 PCB

Este contiene muchas piezas de información asociadas con un proceso especifico, incluyendo:

- *Estado del proceso:* el cual puede ser nuevo, listo, corriendo, esperando, parado, etc.
- *Contador del programa:* El cual indica la dirección de la siguiente instrucción a ser ejecutada por el proceso.
- *Registros del CPU:* el cual varían en número y tipo, dependiendo de la arquitectura de la computadora. Incluyen acumuladores, registros índices, punteros de pilas, y registros de propósito general. Junto con el contador del programa, esta información se debe almacenar cuando ocurre una interrupción, para permitir que el proceso continúe correctamente cuando finalice la interrupción (Figura 4.3).

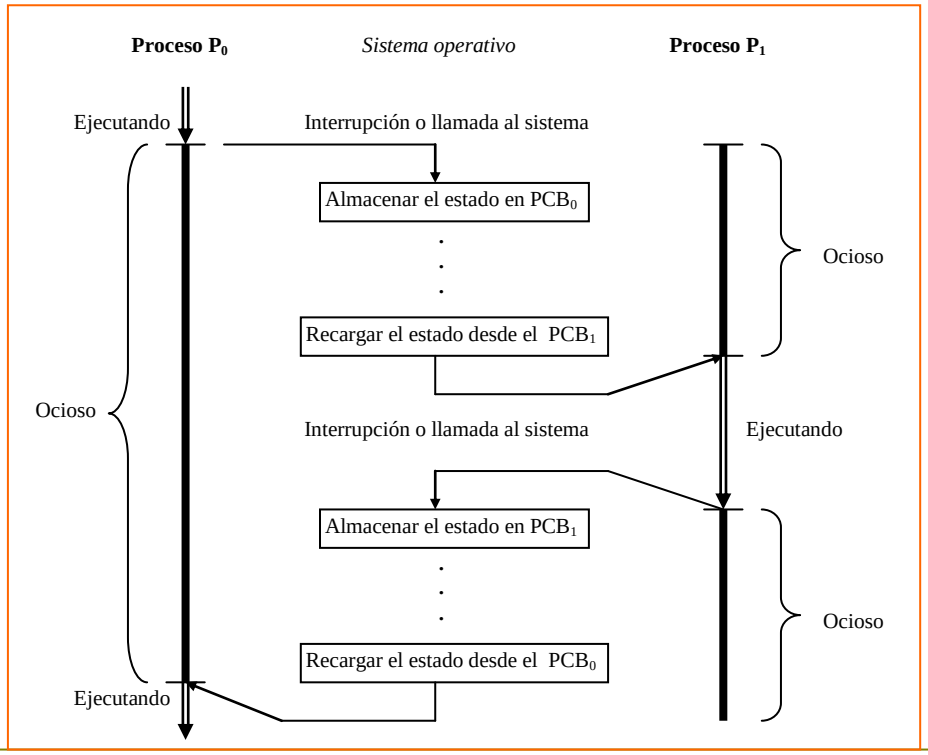


Figura 4.3 Diagrama que muestra el cambio de la CPU desde un proceso a otro.

- *Información del scheduling de la CPU:* el cual incluye la prioridad del proceso, punteros de colas de scheduling, y otros parámetros de scheduling.
- *Información de la administración de memoria:* incluye la información como puede ser el contenido de los registros limite y base, la tabla de páginas, o la tabla de segmentos, dependiendo del sistema de memoria utilizado por el sistema operativo.
- *Información de cantidades:* incluye la cantidad de CPU y tiempo que fue usada, tiempo limites, número de trabajos y procesos, etc.
- *Información de estado de I/O:* la información incluye la lista de dispositivos de I/O (tal como drives de cinta) asignados a este proceso, una lista de archivos abiertos, etc.

El PCB sería el lugar donde se almacena la información que puede variar de proceso en proceso.

Scheduling de procesos

El objetivo de la multiprogramación es el de tener algún proceso corriendo en todo momento, para maximizar la utilización de la CPU. El objetivo del tiempo compartido es el de intercambiar la CPU entre varios procesos tan frecuentemente de manera que los usuarios puedan interactuar con los programas a medida que éstos están corriendo. Para un sistema uniprocesador, allí nunca habrá más que un proceso corriendo en un instante dado. En caso de que halla más procesos, deberán esperar hasta que la CPU se libere y puedan ser elegidos para ocuparla.

Cola de scheduling: Cuando un proceso entra al sistema, es ubicado en una cola de trabajos. Esta cola consiste de todos los procesos en el sistema. Los procesos que están residiendo en la memoria principal y están listos y esperan para ejecutarse son ubicados en una cola llamada cola de listos. Esta cola es generalmente almacenada en una lista vinculada. La cabeza de la cola de listos contendrá un puntero al primer y último PCB de la lista. Cada PCB tiene un puntero al siguiente PCB de la lista.
Existen también otras colas en el sistema. Cuando a un proceso se le asigna la CPU, éste ejecuta por un tiempo y luego abandona, es interrumpido, o espera por la ocurrencia de un evento en particular, tal como

la completitud de un pedido de I/O. En caso que sea un pedido de I/O, tal pedido puede ser de un drive de cinta dedicado (es decir, no compartido), o de un dispositivo compartido, tal como un disco. Ya que existen muchos procesos en el sistema, el disco puede estar ocupado con el pedido de I/O de otro proceso. De esta manera, el proceso puede tener que esperar por el disco. Esta lista de procesos esperando por dispositivo de I/O particular es llamada cola de dispositivos. Cada dispositivo tiene su propia cola de dispositivo.
En cada cola lo que existe es el PCB del proceso en particular y cada cabeza de la cola tiene un puntero al primer y último PCB. De la misma forma, cada PCB tiene un puntero al siguiente PCB.
Una representación común para el proceso de scheduling es un diagrama de cola, como el que se ve en la figura 4.5. Cada rectángulo es una cola. Están presentes dos tipos de colas: la cola de listos y un conjunto de colas de los dispositivos. Los círculos representan los recursos que sirven a las colas, y las flechas indican el flujo del proceso en el sistema.

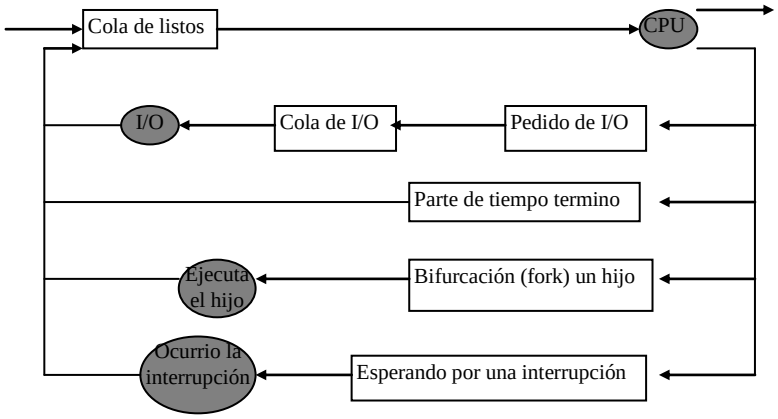


Figura 4.5 Representación del diagrama de encolado de procesos.

Un nuevo proceso es inicialmente puesto en la cola de listos. Este espera en la cola de listos hasta que sea seleccionado para su ejecución (o dispatched) y se le otorga la CPU. Una vez que se le asigna la CPU y es ejecutado, pueden ocurrir uno de varios eventos:

- El proceso podría emitir un pedido de I/O, entonces es ubicado en una cola de I/O.
- El proceso podría crear nuevos subprocesos y esperar por su terminación.
- El proceso podría ser removido forzosamente de la CPU como el resultado de la interrupción, y podría ser puesto otra vez en la cola de listos.

En los primeros dos casos, el proceso cambia de un estado de espera a un estado de listo, por lo que es puesto nuevamente en la cola de listos. El proceso continúa el ciclo hasta que termina, donde se produce la eliminación de su PCB de todas las colas y se liberan los recursos que utilizo.

Schedulers: Un proceso va pasando entre las varias colas de schedulers en su tiempo de vida. El sistema operativo debe seleccionar procesos de estas colas en cada momento. Este proceso de selección es llevado a cabo por el *scheduler*.
En un sistema de lotes, existen a menudo muchos procesos que pueden ser ejecutados inmediatamente. Estos procesos son spooled a un dispositivo de almacenamiento masivo (típicamente un disco), donde son mantenidos para su posterior ejecución. El scheduler de plazo extenso (job scheduler) selecciona procesos de esta "pileta" y los cargan en la memoria principal para que sean ejecutados. El scheduler de plazo pequeño (CPU scheduler) selecciona uno de los diferentes procesos que están listos para ser ejecutados, y le asigna la CPU.
La distinción más importante entre éstos dos schedulers es la frecuencia de su ejecución. El scheduler de corto plazo debe seleccionar un proceso para la CPU muy frecuentemente. Un proceso puede ejecutar solo por unos pocos milisegundos antes de que tenga que esperar por una I/O. El scheduler de largo plazo, por

otro lado, se ejecuta menos frecuentemente. Pueden pasar minutos hasta que se cree un nuevo proceso en el sistema. El scheduler de largo plazo controla el grado de multiprogramación (el número de procesos en memoria). Si el grado de multiprogramación es estable, entonces la tasa promedio de procesos que ingresan en el sistema debe ser igual a la tasa promedio de procesos que dejan el sistema. Así, el scheduler de largo plazo puede necesitar ser invocado solo en caso que un proceso deja el sistema. Por el largo tiempo entre invocaciones del scheduler de largo plazo, se le otorga más tiempo en decidir cual es el próximo proceso que entrará en memoria.

Es importante que el scheduler de largo plazo haga una selección cuidadosa. En general, la mayoría de los procesos pueden ser descriptos como I/O bound (limitado a I/O), o CPU bound (limitado a "ejecutar"). Un proceso I/O bound es aquel que gasta la mayor parte del tiempo haciendo I/O que en cálculos. Un proceso CPU bound es, por otro lado, aquel que genera I/O infrecuentemente, utilizando la mayor parte de su tiempo realizando cálculos. Es importante que el scheduler de largo plazo haga una selección paraje entre los procesos I/O bound y CPU bound, ya que si todos los procesos son I/O bound, la cola de listos estará casi siempre vacía, y el scheduler de corto plazo tendrá poco por hacer. En cambio, si todos los procesos son CPU bound, la cola de espera por I/O estará vacía, los dispositivos estarán sin uso, y el sistema estar nuevamente desbalanceado. El sistema con el mejor rendimiento tendrá una combinación de procesos CPU bound y I/O bound.

En algunos sistemas, el scheduler de largo plazo puede estar ausente o ser mínimo. Por ejemplo, los sistemas de tiempo compartido a menudo no tienen scheduler de largo plazo, ya que ponen cada nuevo proceso en memoria por medio del scheduler de corto plazo. La estabilidad de este sistema depende o de una limitación física (tal como el número de terminales disponibles) o por el propio ajuste de los usuarios humanos. En caso de que la performance decline a niveles inaceptables, algunos usuarios abandonarán.

Algunos sistemas operativos, tal como los sistemas de tiempo compartido, pueden introducir un adicional e intermedio nivel de scheduler. Este scheduler de medio plazo se ve en la figura 4.6. La idea clave del scheduler de medio plazo es que éste puede ser ventajoso para eliminar procesos de la memoria y así reducir el grado de multiprogramación. Un tiempo más tarde, el proceso puede ser reintroducido en la memoria y continuar su ejecución donde se había dejado. Este esquema es llamado *swaping*. El proceso es swap in y swap out por el scheduler de medio plazo. Swaping puede ser necesario para mejorar la mezcla de procesos, o por un cambio en los requerimientos de la memoria el cual pueden provocar que se disminuya la cantidad de memoria disponible, teniendo que liberar memoria.

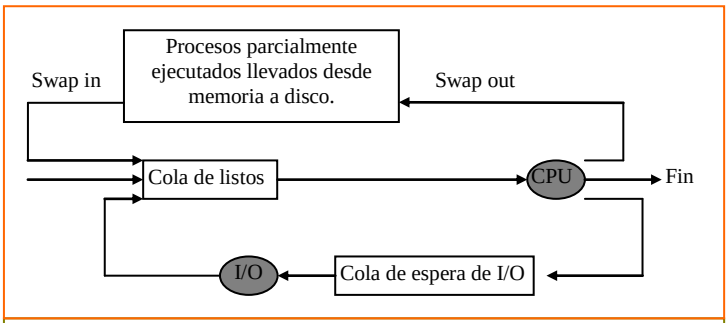


Figura 4.6 Agregado de un scheduler de medio plazo.

Context switch: Cambiar la CPU a otro proceso requiere guardar el estado del proceso viejo y cargar el estado salvado del nuevo proceso. Esta tarea es conocida como *context switch*. El tiempo del context switch es puro overhead, ya que el sistema no hace trabajo útil mientras cambia de procesos. Su velocidad varia según la maquina, dependiendo de la velocidad de memoria, el número de registros que debe copiar, y la existencia de instrucciones especiales (tal como una única instrucción que almacena o carga todos los registros).

Operación sobre procesos

Los procesos en un sistema pueden ejecutar concurrentemente, y deben ser creados y eliminados dinámicamente. Así, el sistema operativo debe proveer mecanismos para la creación y terminación de procesos.

Creación de procesos: Un proceso puede crear nuevos varios procesos, vía una llamada al sistema *create-process*, durante el curso de su ejecución. El proceso creador es el proceso padre, mientras que los nuevos procesos son hijos de este proceso. Cada uno de estos nuevos procesos puede, a su vez, crear otros nuevos procesos, formando un árbol de procesos.
En general, un proceso necesitara ciertos recursos (tiempo de CPU, memoria, archivos, dispositivos de I/O) para llevar a cabo su tarea. Cuando un proceso crea un subproceso, el subproceso puede ser capaz de obtener sus recursos directamente desde el sistema operativo, o puede ser restringido a un subconjunto de los recursos del proceso padre. El padre puede tener particionados sus recursos entre sus hijos, o puede ser capaz de compartir algunos recursos (tal como memoria o archivos) entre varios de sus hijos. Con la restricción de que los recursos de un proceso hijo sean una parte de los recursos del proceso padre previene a cualquier proceso sobrecargar el sistema por la creación de muchos subprocesos.
Además, de los varios recursos físicos y lógicos que un proceso obtiene en el momento que es creado, la inicialización de los datos (entrada) puede ser pasada por el proceso padre al proceso hijo. Por ejemplo, consideremos un proceso en el cual su función es mostrar el estado de un archivo, digamos F1, sobre la pantalla de una terminal. Cuando este proceso es creado, éste conseguirá como entrada, por medio de su proceso padre, el nombre del archivo F1 y ejecutara a partir de este dato para obtener la información deseada. Este puede obtener el nombre del dispositivo de salida. Algunos sistemas operativos pasan recursos al proceso hijo. En tales sistemas, el nuevo proceso puede obtener dos archivos abiertos, F1 y el dispositivo de destino, por lo que solo necesita transferir la información entre ambos.
Cuando un proceso crea un nuevo proceso, dos posibilidades existen en términos de ejecución:

- El padre continúa la ejecución concurrentemente con su hijo.
- El padre espera hasta que algunos o todos sus hijos hayan terminado.

Existen también dos posibilidades en términos del espacio de direcciones del nuevo proceso:

- El proceso hijo es un duplicado del proceso padre.
- El proceso hijo tiene un programa cargado en él.

Para ilustrar estas diferentes implementaciones, consideremos el sistema operativo UNIX. En UNIX, cada proceso es identificado por su identificador de proceso, el cual es un entero único. Un nuevo proceso es creado por la llamada al sistema **fork**. El nuevo proceso consiste de una copia del espacio de direcciones del proceso original (el **fork** crea un duplicado exacto del proceso original, incluidos todos los descriptores de archivos, registros, ..., todo). Este mecanismo permite al proceso padre comunicarse fácilmente con sus hijos. Ambos procesos (el padre y el hijo) continúan la ejecución de la siguiente instrucción luego del **fork** con una diferencia: el código de retorno del **fork** es cero para el nuevo (hijo) proceso, mientras que el identificador del proceso del hijo es retornado al padre (un valor distinto de cero).
Típicamente, la llamada al sistema **execve** es usada luego del **fork** por uno de estos dos procesos para reemplazar el espacio de memoria del proceso con un nuevo programa. La llamada al sistema **execve** carga un archivo binario en la memoria (destrozando la imagen de memoria del programa que contiene la llamada al sistema **execve**) y comienza su ejecución. De esta manera, los dos procesos son capaces de comunicarse, y luego continuar de manera separada. El padre puede entonces crear nuevos hijos, o en caso de que no tenga nada que hacer mientras su hijo corre, puede utilizar una llamada al sistema **wait** para moverse a sí mismo desde la cola de listos hasta que el hijo termine su ejecución.
Entonces, luego del fork, el proceso original y la copia (el padre y el hijo) siguen cada quien su camino. Todas las variables tienen valores idénticos luego del fork, pero dado que los datos del padre se copian para crear el hijo, los cambio subsecuentes en uno de ellos no afectará al otro (el área de texto, que es

inalterable, es compartido entre el padre y el hijo). La llamada al fork devuelve un valor que es cero en el hijo, e igual al identificador del proceso hijo en el proceso. Así, entre ellos pueden saber cual es el hijo y cual el padre. En la mayoría de las veces, luego del fork el hijo debe ejecutar código diferente al del padre. Por ejemplo, en el caso de un shell: éste lee un comando de la terminal, hace un fork para crear un proceso hijo, espera que el hijo ejecute el comando, y al terminar el hijo, lee el siguiente comando. Para esperar al que el hijo termine, el padre ejecuta una llamada al sistema **wait** que hace que espere hasta que el hijo termine. A continuación se presenta un ejemplo simple de un shell:

```
while (TRUE){                                   // Repetir siempre.
        read_command(command, parameters)       // Leer entrada de la terminal.
        if (fork() != 0){                       // Crear y separar proceso hijo .
                // Código del Padre
                wait;                           // Esperar que el hijo salga.
        }else{
                // Código del hijo
                execve;                         // Ejecutar comando.
        }
}
```

Shell reducido al mínimo.

El sistema operativo DEC VMS, en contraste, crea un nuevo proceso, carga un programa especifico en el espacio de direcciones de éste proceso, y comienza su ejecución. El sistema operativo Microsoft NT soporta ambos modelos: el espacio de direcciones del padre puede ser duplicado, o el padre puede especificar el nombre de un programa al sistema operativo para ser cargado en el espacio de direcciones del nuevo proceso.

Terminación de procesos: Un proceso termina cuando finaliza su ejecución y envía al sistema operativo una llamada de sistema **exit** para eliminarse. En este punto, el proceso puede retornar datos a su proceso padre (vía la llamada de sistema **wait**). Todos los recursos del sistema, incluyendo memoria virtual y física, archivos abiertos, buffers de I/O, son liberados por el sistema operativo.
Existen circunstancias adicionales cuando la terminación ocurre. Un proceso puede causar la terminación de otro proceso por medio de una apropiada llamada de sistema (por ejemplo, **abort**). Usualmente, tales llamadas de sistema pueden ser invocadas solo por el proceso padre del proceso que será terminado. Sino, los usuarios matarían entre ellos sus trabajos. Note que un proceso padre necesita conocer el identificador de sus procesos hijos. Así, cuando un proceso crea un nuevo proceso, el identificador del nuevo proceso creado es pasado al padre.
Un padre puede terminar la ejecución de alguno de sus hijos por varias razones, tales como:

- El hijo a excedido el uso de alguno de sus recursos que se le habían asignado.
- La tarea para el cual se creo el hijo no se necesita.
- El padre terminó, y el sistema operativo no permite que un hijo continúe en caso de que su padre haya terminado.

Para determinar el primer caso, el padre debe tener un mecanismo para inspeccionar el estado de sus hijos. Muchos sistemas, incluyendo VMS, no permiten que un hijo exista en caso de que su padre haya terminado. En tales sistemas, si un proceso termina (ya sea normalmente o anormalmente), entonces todos sus hijos deben terminar también. Esto es conocido como terminación en cascada y es normalmente realizado por el sistema operativo.
Para ilustrar la ejecución y terminación de procesos, consideremos nuevamente el sistema operativo UNIX. En UNIX, un proceso puede terminar usando la llamada al sistema **exit**, y su proceso padre puede esperar por el evento utilizando la llamada al sistema **wait**. La llamada al sistema **wait** retorna el identificador del proceso de un hijo terminado, por lo que el padre puede decir cual de sus hijos ha terminado. Sin embargo, en caso de que el padre termine, todos sus hijos son terminados por el sistema operativo. Sin un padre, UNIX no sabe a quien dar la información de las actividades de un hijo.

Cooperación entre procesos

Un proceso que esta siendo ejecutado en el sistema operativo puede ser un proceso independiente o un proceso de cooperación. Un proceso independiente es aquel que no puede afectar ni ser afectado por ningún otro proceso que esta ejecutando en el sistema. Claramente, un proceso que no comparte ningún dato con ningún proceso es independiente. Por otro lado, un proceso esta cooperando si éste puede afectar o ser afectado por otros procesos ejecutando en el sistema. Claramente se ve que un proceso que comparte datos con otro proceso es un proceso de cooperación.

Existen varias razones por el cual se desea proveer un entorno de cooperación entre procesos:

- *Compartir información:* Ya que varios usuarios pueden estar interesados en la misma pieza de información (por ejemplo, un archivo compartido), debemos proveer un entorno que permita el acceso concurrente para estos tipos de recursos.
- *Aumento de velocidad de computo:* Si queremos que una tarea particular corra más rápido, debemos romperla en varias subtareas, cada una de las cuales podría ejecutar en paralelo con las demás. Note que tal aumento de velocidad puede ser conseguida en caso de que la computadora tenga múltiples elementos de procesamiento (tales como CPUs o canales de I/O).
- *Modularidad:* se desea que el sistema sea dividido en módulos, dividiendo las funciones del sistema en procesos separados, como se vio en el capítulo 3.
- *Conveniencia:* Puede que un único usuario tenga muchas tareas para trabajar en un tiempo. Por ejemplo, un usuario puede estar editando, imprimiendo, y compilando en paralelo.

La ejecución concurrente que requiere la cooperación entre procesos, requiere mecanismos que le permitan la comunicación entre procesos y la sincronización de sus tareas.

Para ilustrar el concepto de cooperación de procesos, consideremos el problema productor-consumidor. Un proceso productor produce información que es consumida por un proceso consumidor. Por ejemplo, un programa de impresión produce caracteres que son consumidos por el driver de la impresora. Un compilador puede producir código asembler, el cual es consumido por un asembler. El asembler, de hecho, puede producir módulos objeto, los cuales son consumidos por el cargador.

Para permitir a los procesos productor y consumidor correr concurrentemente, se debe tener un buffer de ítems que puede ser llenado por el productor y vaciado por el consumidor. Un productor puede producir un ítem mientras que el consumidor esta consumiendo otro ítem. El productor y el consumidor deben estar sincronizados, para que el consumidor no trate de consumir un ítem que aún no ha sido creado. En esta situación, el consumidor debe esperar hasta que se produzca un ítem.

El buffer puede estar proveído por el sistema operativo por medio del uso de un IPC, o proveído explícitamente por el programa con el uso de la memoria compartida (en este caso, el programador debe generar código que soporte tal memoria compartida). Ilustraremos una solución de memoria compartida para el problema del buffer con limites, es decir, en caso de que no haya ítems debe esperar el consumidor, en caso de que no haya lugar en el buffer debe esperar el productor. El proceso productor y consumidor comparten las siguientes variables:

```
var n;
type ítem = ...;
var buffer: array [0..n-1] of ítem;
        in, out: 0..n-1;
```

con las variables in y out inicializadas con el valor 0. El buffer compartido es implementado como un arreglo circular con dos punteros lógicos: in y out. La variable in apunta a la siguiente posición libre en el buffer; la variable out apunta a la primera posición llena del buffer. El buffer esta vacío cuando in = out; el buffer esta lleno cuando in + 1 **mod** n = out.

Los códigos de los procesos productor y consumidor son los siguientes. La instrucción *no-op* es la instrucción que representa hacer nada. Así, **while** condición **do** do-op simplemente testea la condición repetitivamente hasta que se convierta en falsa.

El proceso productor tiene una variable local *nextp*, en el cual se almacena el nuevo ítem a ser producido:

```
repeat
        ...
        producir un ítem en nextp
        ...
        while in + 1 mod n = out do no-op;
        buffer[in] := nextp;
        in := in + 1 mod n;
until false;
```

El proceso consumidor tiene una variable local *nextc*, en el cual se almacena el ítem a ser consumido:

```
repeat
        while in = out do no-op;
        nextc = buffer[out];
        out := out + 1 mod n;
        ...
        consumir el ítem en nextc
        ...
until false;
```

Este esquema permite que haya en el buffer como mucho n-1 ítems a la vez.

Threads

Recalcamos que un proceso esta definido por los recursos que éste usa, y por el lugar donde es ejecutado. Existen muchas instancias, sin embargo, en el cual seria muy útil que los recursos sean compartidos y accedidos concurrentemente. Esta situación es similar en el caso que una llamada al sistema **fork** es invocada con un nuevo contador de programa, o thread de control, ejecutando en el mismo espacio de direcciones.

Un ejemplo donde los threads son muy útiles es el caso de los navegadores de la WW tal como Netscape o Mosaic. Muchas páginas web tienen múltiples imágenes pequeñas. Para cada imagen de una página web, el navegador debe establecer una conexión individual con el sitio de la página de casa y solicitar la imagen. Se desperdicia una gran cantidad de tiempo estableciendo y liberando éstas conexiones. Si tenemos múltiples threads dentro del navegador, podemos solicitar muchas imágenes al mismo tiempo, acelerando considerablemente el rendimiento en la mayor parte de los casos, ya que en el caso de imágenes pequeñas, el tiempo de preparación es el factor determinante, no la rapidez de la línea de transmisión.

En algunos sistemas, el sistema operativo no esta consciente de la existencia de los threads. Es decir, los threads se manejan a nivel de usuario. Por ejemplo, cuando un threads está a punto de bloquearse, escoge e inicia a su sucesor antes de detenerse. Existen varios paquetes de threads a nivel de usuario (ejemplo, threads P de Posix, threads C de Mach).

En otros sistemas, el sistema operativo está consciente de la existencia de múltiples threads por proceso, por lo que, al bloquearse un thread, el sistema operativo escoge el que se ejecutará a continuación, ya sea del mismo proceso o de otro. Para realizar esta planificación (scheduling), el kernel debe tener una tabla de threads que liste todos los threads del sistema, análoga a la tabla de procesos.

Aunque dichas alternativas pueden parecer equivalentes, el rendimiento es muy distinto. El cambio de threads es mucho más rápido cuando la administración de los threads se efectúa a nivel de usuario que cuando se ejecuta con una llamada al kernel (threads a nivel del kernel). Este hecho nos diría que es mejor realizar los threads a nivel de usuario. Por otro lado, cuando los threads se manejan a nivel de usuario y uno se bloquea, el kernel bloquea todo el proceso, ya que no sabe de la existencia de los threads. Este hecho nos diría que es mejor realizar threads a nivel del kernel. La consecuencia es que se usan ambos sistemas y hasta algunos híbridos.

Estructura del thread: Un thread (o proceso de peso liviano) es una unidad básica de la utilización de la CPU, y consiste de un contador de programa, un conjunto de registros, y un espacio de pila. Éste comparte con threads pares la sección de código, la sección de datos, y los recursos del sistema operativo tales como los archivos abiertos y las señales, colectivamente conocido como una tarea. Un proceso tradicional o proceso pesado seria una tarea con solo un thread. Una tarea no hace nada si no hay un thread en ella, y un thread puede estar solo en una tarea. Esta extensión de la compartición de los recursos hace que el switching de la CPU a través de threads pares y la creación de threads sea mucho menos "cara", comparada con el context switch a través de procesos pesados. Aunque un thread context switch aun requiere el cambio de un conjunto de registros, no se necesita hacer trabajo con respecto a la administración de la memoria. Como un entorno de procesamiento en paralelo, un proceso con muchos threads puede introducir problemas de control de concurrencia que requieren el uso de secciones criticas o trabas.

Algunos sistemas también implementan threads a nivel del usuario en librerías a nivel usuario, en lugar de llamadas de sistema, por lo que el switcheo de threads no necesita llamar al sistema operativo, y causar ante esto una interrupción al kernel. Ante esto, el switcheo se puede hacer más rápido. Sin embargo, los threads a nivel de usuario tienen sus desventajas. Por ejemplo, si el kernel es de único threads, entonces cualquier threads a nivel de usuario que ejecute una llamada al sistema causará que la tarea completa deba esperar hasta que la llamada al sistema retorne.

Podemos comprender la funcionalidad de los threads comparando el control de multi-threads con el control de multi-proceso. Con múltiples procesos, cada proceso opera independientemente de los otros; cada proceso tiene su propio contador de programa, registro de pila y espacio de direcciones. Este tipo de organización es muy útil cuando los trabajos realizados por los procesos no están relacionados. Pero múltiples procesos pueden realizar también la misma tarea. Por ejemplo, múltiples procesos pueden proveer datos a una maquina remota. Sin embargo, es más eficiente para ésta tarea tener un solo proceso sosteniendo múltiples threads sirviendo para el mismo propósito. En la implementación de múltiples procesos, cada proceso ejecuta el mismo código, pero tiene su propia memoria y recursos de archivos. Un proceso con muchos threads usa menos recursos que múltiple procesos, ya que comparten memoria, archivos abiertos y scheduling de la CPU. Por ejemplo, como Solaris desarrolla, los trabajos de redes están siendo reescritos como threads en el kernel para incrementar el rendimiento de las funciones que sirven a las funciones de red.

Los threads en gran parte operan como lo hacen los procesos: pueden estar en uno de varios estados: listo, bloqueado, corriendo, terminado, etc, comparten la CPU, y solo un thread a la vez esta activo (corriendo). Un thread en un proceso ejecuta secuencialmente, y cada thread tiene su pila y contador de programa. Los thread pueden crear thread hijos, y pueden bloquearse esperando por una llamada al sistema que indica la completitud del hijo. Si un thread esta bloqueado, entonces otro esta corriendo. Sin embargo, en contraposición que los procesos, los threads no son independientes uno de otros. Ya que cada thread puede acceder a cualquier dirección de la tarea, un thread puede escribir o leer sobre cualquier otra pila de otro thread. Sin embargo, esta estructura no provee protección entre threads. Tal protección no debería ser necesaria: mientras los procesos pueden ser originados por diferentes usuarios, y pueden ser enemigos entre sí, solo un único usuario puede ser dueño de una tarea con múltiples threads. Los threads, en este caso, probablemente serán diseñados para asistirse entre sí, y de esta forma no se requiere protección mutua. En la figura 4.8 se ve una tarea con múltiples threads.

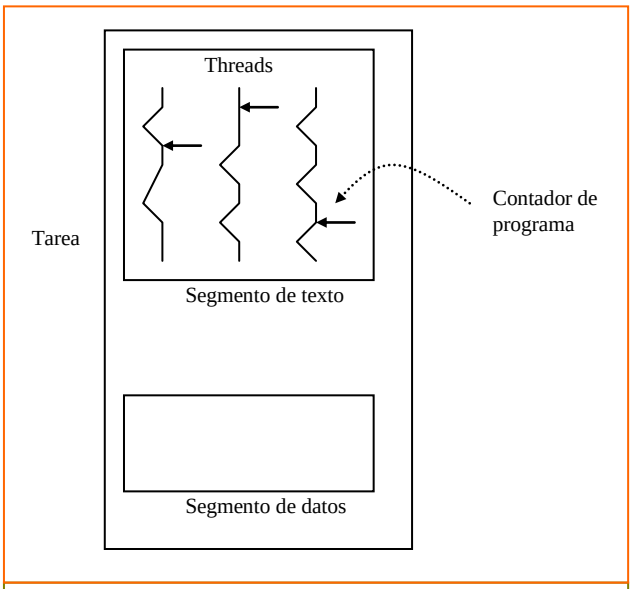


Figura 4.8 Múltiple threads en una tarea.

Retornemos a nuestro ejemplo del proceso bloqueado del servidor de archivos en el modelo de un único proceso. En este entorno, ningún otro proceso de servidor puede ser ejecutado hasta que el primer proceso se desbloquee. Por el contrario, en el caso de una tarea con múltiples threads, mientras que un threads de servidor esta bloqueado y esperando, un segundo thread en la misma tarea podría correr. En esta aplicación, la cooperación de múltiples threads, los cuales son parte de una misma tarea, otorga la ventaja de aumentar el rendimiento. Otra aplicación, tal como el problema del productor-consumidor, requiere compartir un buffer en común por lo que también seria útil la utilización de threads: el productor y el consumidor podrían ser threads en una tarea. Se necesita poco overhead para cambiar entre threads, y, en un sistema multiproceso, ellos podrían ejecutar en paralelo en dos procesadores para máxima eficiencia.

Los threads proveen un mecanismo que permite a procesos secuenciales hacer llamadas al sistema de bloqueos, mientras que también se consigue paralelismo. Para ilustrar la ventaja de este mecanismo, consideremos escribir un servidor de archivos en un sistema donde no están disponibles los threads. Ya hemos visto que, en un servidor de archivo de único thread, el proceso servidor debe llevar a cabo un pedido para completarse antes de adquirir un nuevo trabajo. En caso de que el pedido involucre esperar por acceso a disco, la CPU estará ociosa durante la espera. De hecho, el número de pedidos por segundo que pueden ser procesados es mucho menor que con la ejecución en paralelo. Sin la opción de múltiple threads, un diseñador de sistema buscaría minimizar la baja performance de los procesos con un único threads por medio de la imitación de la estructura en paralelo de los threads usando procesos pesados. Se podría hacer, pero seria muy costos y seria una estructura de programa no secuencial.

La abstracción presentada por un grupo de procesos livianos es la de control de múltiple threads asociados con varios recursos compartidos. Existen muchas alternativas considerando threads. Los threads pueden ser soportados por el kernel. En este caso, se provee un conjunto de llamadas de sistema similares para las de procesos. Alternativamente, los threads pueden ser soportados por encima del kernel, vía un conjunto de llamadas a librerías a nivel del usuario.

Los threads a nivel del usuario no involucran el kernel, ante esto son más rápidos para cambiar que los threads que deben ser cambiados pasando por el kernel. Sin embargo, cualquier llamada al sistema operativo puede causar que el proceso entero deba esperar, ya que el kernel trabaja a nivel de procesos (sin el conocimiento de la existencia de threads para hacer el sheduling), y un proceso que esta esperando no tiene tiempo de CPU. El scheduler puede ser también injusto. Consideremos dos procesos, unos con un thread (proceso *a*) y el otro con 100 threads (proceso *b*). Cada proceso generalmente recibe la misma cantidad de tiempo de la CPU, por lo que el thread en el proceso *a* correrá 100 veces más rápido que un thread en el proceso *b*. Un sistema donde el kernel soporte threads, el switcheo a través de los threads

llevaría a consumir más tiempo ya que el kernel (vía una interrupción) llevaría a cabo el switcheo de los threads. Sin embargo, cada thread podría ser scheduled independientemente, por lo que el proceso *b* recibiría 100 veces el tiempo de la CPU mientras que el proceso *a* lo recibiría una vez. Adicionalmente, el proceso *b* podría tener 100 llamadas de sistema en concurrente operación, logrando mucho más de lo que lograría el mismo proceso en un sistema que soporte solo threads a nivel de usuario.
Por los compromisos que involucran cada uno de estos modelos de threads, algunos sistemas utilizan un híbrido en el cual es implementado tanto los threads a nivel de usuario como a nivel del kernel. Uno de esos sistemas es Solaris 2.
Los threads están ganando en popularidad ya que ellos tienen alguna de las características de procesos pesados pero pueden ejecutar concurrentemente. Existen muchas aplicaciones donde ésta combinación es muy útil. Por ejemplo, algunas implementaciones del kernel de UNIX son de única tarea: solo una tarea puede estar ejecutando código en el kernel a la vez. Muchos problemas, tal como la sincronización de accesos a los datos (trabando las estructuras de datos mientras están siendo modificadas) son eliminadas, ya que solo un proceso puede conseguir hacer la modificación. Otros sistemas (el de Mach) son multithreads, permitiendo al kernel servir muchos pedidos simultáneamente. En este caso, los threads por si mismos son sincrónicos: otro thread en el mismo grupo puede correr solo si el thread que esta ejecutando actualmente abandona el control. Por supuesto, el thread actual abandonara el control solo cuando este no estaba modificando datos compartidos. En los sistemas donde los threads son asincrónicos, algunos mecanismos de trabas explícitos deben ser utilizados, así como en sistemas donde múltiples procesos comparten datos.

Ejemplo: Solaris 2. Veamos un ejemplo en el uso de threads en un sistema operativo: Solaris 2, una versión de UNIX, que ha sido transformada para soportar threads a nivel de usuario y de kernel, multiprocesomiento simétrico y scheduling en tiempo real.
Solaris 2 soporta threads a nivel de usuario como se vio anteriormente. Ellos son soportados por una librería para su creación y scheduling, y el kernel no conoce nada de estos threads. Solaris 2 espera tener miles de threads a nivel de usuario compitiendo por ciclos de CPU.
Solaris 2 también define un nivel intermedio de threads. Entre los threads a nivel de usuario y a nivel de kernel están los procesos livianos (LWP). Cada tarea (proceso) contiene por lo menos un LWP. Estos LWP están manipulados por la librería de threads. Los threads a nivel de usuario están conectados sobre los LWP que tiene cada proceso, y sólo los threads a nivel de usuario que están actualmente conectados con el LWP consiguen trabajar. El resto de los threads están o bloqueados o esperando por un LWP en el cual puedan correr.
Todas las operaciones en el kernel son ejecutadas por threads a nivel de kernel estándar. Hay un thread a nivel de kernel para cada LWP, y hay algunos threads a nivel de kernel el cual corren en nombre del kernel y no tienen asociado un LWP (por ejemplo, un thread para servir los pedidos de disco). El sistema completo se ve en la figura 4.9. Los threads a nivel de kernel son los únicos objetos que son scheduled en el sistema. Algunos threads a nivel de kernel están unidos con los procesadores en el sistema, mientras que algunos están atados a un procesador especifico. Por ejemplo, el thread del kernel asociado con un driver de dispositivo para un dispositivo conectado a un procesador especifico correrá solo sobre éste procesador. Por demanda, un thread puede también ser fijado a un procesador. Solo éste thread correrá sobre el procesador, con el procesador asignado solo a éste thread (ver el thread de más a la derecha en la figura 4.9).

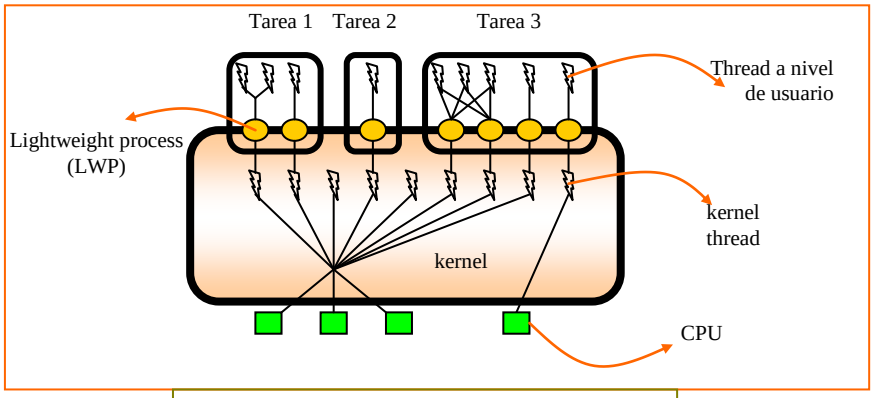


Figura 4.9 Threads en Solaris 2.

Consideremos este sistema en operación. Cualquier tarea puede tener muchos threads a nivel de usuario. Estos threads a nivel de usuario pueden ser scheduled y cambiados entre los diferentes LWP para el proceso sin la intervención del kernel. No se necesita context switch para un thread a nivel de usuario para bloquear a uno y comenzar a correr otro, por lo que los threads a nivel de usuario son extremadamente eficientes.

Estos threads a nivel de usuario son soportados por los LWP. Cada LWP es conectado a exactamente un thread a nivel de kernel, mientras que cada thread a nivel de usuario es independiente del kernel (el kernel ni sabe que existe). Puede haber muchos LWP en una tarea, pero sólo se necesitan cuando los thread se necesitan comunicar con el kernel. Por ejemplo, consideremos 5 pedidos diferentes de lectura de archivo que ocurren simultáneamente. Entonces, se necesitarían 5 LWP, ya que todos ellos podrían estar esperando por la completitud de la I/O en el kernel. Si una tarea sólo tiene 4 LWP, entonces el quinto pedido tendrá que esperar hasta que uno de los LWP retorne desde el kernel. Agregar un sexto LWP podría no ganar nada en caso de que ya haya bastante trabajo con lo 5 existentes.

Los threads del kernel son planificados (scheduler) por el scheduler del kernel y ejecutados en la CPU o CPUs del sistema. Si el thread del kernel se bloquea (usualmente esperando por la completitud de una I/O), el procesador esta libre de correr otro thread del kernel. Si el thread que se bloqueo estaba corriendo en nombre de un LWP, el LWP también se bloquea. Continuando con la cadena, el thread a nivel de usuario actualmente vinculado con el LWP también se bloquea. Si la tarea conteniendo este thread solo tiene un LWP, la tarea completa se bloquea hasta que se complete la I/O. Este comportamiento es el mismo que el de un proceso en la antigua versión del sistema operativo.

Con Solaris 2, una tarea no debe bloquearse por mucho tiempo en espera de la completitud de una I/O. La tarea puede tener múltiples LWP; si uno se bloquea, los otros puede continuar su ejecución en la tarea.

Se terminara este ejemplo analizando los recursos necesarios para cada uno de estos tipos de threads.

- Un thread de kernel solo tiene una pequeña estructura de datos y una pila. El cambio entre los threads del kernel no requieren cambiar la información de acceso a memoria, y, por esta causa, el cambio es rápido.
- Un LWP contiene un bloque de control de proceso con registros de datos, información de conteo, y información de memoria. El cambio entre los LWPs es, por lo tanto, más lento.
- Un thread a nivel de usuario necesita solo una pila y un contador de programa: no se requieren recursos del kernel. El kernel no está involucrado en el scheduling de estos threads a nivel de usuario; por lo tanto, el cambio entre ellos es rápido. En una tarea puede haber cientos de estos threads pero el kernel lo único que verá es el LWP en el proceso que soporta éstos threads a nivel de usuario.

Comunicación entre procesos

Anteriormente se vio como los procesos que cooperan pueden comunicarse en un entorno de memoria compartida. El esquema requiere que éstos procesos compartan un buffer en común, y que el código para implementar el buffer sea explícitamente escrito por el programador de la aplicación. Otra forma de

conseguir el mismo efecto es por medio del sistema operativo, el cual debe proveer las ideas para la comunicación de procesos para que se puedan comunicar entre ellos (IPC).
IPC provee un mecanismo que permite la comunicación entre procesos y la sincronización de sus acciones. La mejor forma de provocar esta comunicación es por medio del sistema de mensajes. Este sistema puede ser definido de varias maneras.
Note que los esquemas de comunicación por medio de la memoria compartida y el sistema de los mensajes no se excluyen mutuamente, y ambos podrían ser usados simultáneamente en un único sistema operativo o aun en un único proceso.

Estructura básica: La función del sistema de mensajes es la de permitir a los procesos comunicarse entre ellos sin la necesidad de recurrir a las variables compartidas. IPC provee al menos dos operaciones: **send** (mensaje) y **receive** (mensaje).
Los mensajes enviados por un proceso pueden ser de tamaño fijo o variable. Si los mensajes enviados solo pueden ser de tamaño fijo, la implementación física es sencilla. Esta restricción, sin embargo, hace las tareas de programación más difíciles. Por otro lado, los mensajes de tamaño variable requieren una implementación física más compleja, pero las tareas de programación se convierten fáciles.
Si un proceso P y otro Q quieren comunicarse se debe formar un enlace de comunicación entre ellos. Este enlace puede ser implementado por una gran variedad de formas. En este capítulo nos centraremos en la implementación lógica. Algunas de estas cuestiones de implementación son:

- ¿Cómo se establece un enlace?
- ¿Puede un enlace ser asociado con más de un proceso?
- ¿Cuántos enlaces pueden existir entre cada par de procesos?
- ¿Qué es la capacidad de un enlace?. Es decir, ¿tiene un enlace espacio de buffer?, si es así, ¿cuántos?.
- ¿Cuál es el tamaño de un mensaje?. ¿Puede un enlace adaptarse a los mensajes de tamaño variable, o solo acepta mensajes de tamaño fijo?
- ¿Un enlace es unidireccional o bidireccional?. Esto es, si existe un enlace entre P y Q, ¿los mensajes pueden ir solo en una dirección o en ambas direcciones?.

Diremos que un enlace es unidireccional sólo si cada proceso conectado al enlace puede enviar o recibir, pero no ambos, y cada enlace tiene al menos un proceso recibidor conectado a él.
Además, existen varios métodos para implementar lógicamente un enlace y las operaciones de **send/receive**:

- Comunicación directa o indirecta.
- Comunicación simétrica o asimétrica.
- Buffering explícito o automático.
- Envíos por copia o por referencia.
- Mensajes de tamaño variable o fijo.

A continuación analizaremos cada uno de estos puntos.

Nombre: Un proceso que quiere comunicarse debe tener una forma de poder diferenciarse de los demás. Ellos pueden usar la comunicación directa o la indirecta.

- ***Comunicación directa:*** en este tipo de comunicación, cada proceso que quiere comunicarse debe explícitamente nombrar el nombre del transmisor o del recibidor. En este esquema, las primitivas **send** y **receive** son definidas como sigue:

  send(P, mensaje). Envía un mensaje al proceso P.
  receive(Q, mensaje). Recibe un mensaje del proceso Q

Un enlace de comunicación con este esquema tiene las siguientes propiedades:

- Un enlace es establecido automáticamente entre todos los pares de procesos que quieren comunicarse. El proceso necesita conocer sólo la identificación del otro para comunicarse.
- Un enlace es establecido con exactamente dos procesos.
- Entre cada par de procesos existe solo un enlace.
- El enlace puede ser unidireccional, pero generalmente es bidireccional.

Para ilustrar el esquema se presenta una solución al problema del productor-consumidor. Se permite que los procesos productor y consumidor corran concurrentemente, por lo que el productor puede estar produciendo un ítem mientras el consumidor esta consumiendo otro. Cuando el productor termina de generar un ítem, éste envía (**send**) el ítem al consumidor. El consumidor obtiene el ítem vía la operación **receive**. En caso de que un ítem no haya sido producido todavía, el proceso consumidor debe esperar hasta que se produzca uno. El proceso productor se define como sigue:

```
repeat
        ...
        producir un ítem en nextp
        ...
        send (consumer, nextp);
until false;
```

El proceso consumidor se define como:

```
repeat
        receive(producer, nextc);
        ...
        consumir el ítem en nextc
        ...
until false;
```

Este esquema exhibe una simetría en el direccionamiento; es decir, ambos procesos consumidor y productor tienen el nombre del otro para comunicarse. Una variante de este esquema emplea asimetría en el direccionamiento. Solo el que envía nombra al recibidor; el que recibe el mensaje no necesita nombrar al transmisor. En este esquema, las primitivas send y receive se definen como:

send(P, mensaje). Envía un mensaje al proceso P.

receive(id, mensaje). Recibe un mensaje de algún proceso; la variable id es puesta con el nombre del proceso con el cual la comunicación tomo lugar.

La desventaja en ambos esquemas (simétrico y asimétrico) es que en caso de que se cambie el nombre de un proceso, entonces se debe examinar la definición de todos los demás procesos. Se deben encontrar todas las referencias al viejo nombre para ser cambiada por el nuevo nombre.

- ***Comunicación indirecta:*** Con este tipo de comunicación, los mensajes son enviados a, y recibidos de buzones (o puertos). Un puerto puede ser visto como un objeto en el cual los mensajes pueden ser ubicados por los procesos y desde donde se pueden sacar los mensajes. Cada puerto tiene una identificación única. En este esquema, un proceso se puede comunicar con algún otro por medio de un conjunto de diferentes puertos. Los dos procesos se pueden comunicar sólo si ellos tienen un puerto compartido. Las primitivas send y receive se definen como:

  send(A, mensaje). Envía un mensaje al puerto A.

  receive(A, mensaje). Recibe un mensaje del puerto A.

En este esquema, un enlace de comunicación tiene las siguientes propiedades:

- El enlace se establece entre pares de procesos sólo si tienen un puerto compartido.
- Un enlace se puede asociar con más de dos procesos.

- Entre cada par de procesos comunicándose, puede haber un número diferente de enlaces, cada uno correspondiente a un puerto.
- En enlace puede ser unidireccional o bidireccional.

Supongamos ahora que los procesos P1, P2 y P3 tienen un puerto compartido A. El proceso P1 envía (**send**) un mensaje a A, mientras que P2 y P3, cada uno de ellos ejecuta una primitiva **receive** de A. ¿Cuál de los dos procesos recibirá el mensaje enviado por P1?. Esta cuestión se puede resolver en una variedad de formas:

- Permitir que un enlace sea asociado con, a lo sumo, dos procesos.
- Permitir que a lo sumo un proceso a la vez ejecute la operación **receive**.
- Permitir que el sistema seleccione arbitrariamente cual proceso recibirá el mensaje (es decir, entre P2 o P3, pero no ambos).

Un puerto puede ser propiedad de un proceso o del sistema. En caso de que sea propiedad de un proceso (es decir, el puerto esta unido a un proceso, o se define como parte del proceso), entonces distinguimos entre el propietario del puerto (aquel que solo puede recibir mensajes del puerto), y los usuarios del puerto (aquellos que solo envían mensajes al puerto). Cuando un proceso que era propietario del puerto termina, entonces el puerto desaparece. Luego de esto, cualquier proceso que le envíe mensajes a este puerto debe ser notificado que el puerto ya no existe (por medio de una excepción).
Existen varias formas para designar el propietario y los usuarios de un puerto. Una posibilidad es la de permitir a un proceso definir variables de tipo *puerto*, con lo que aquel proceso que declare dicha variable será el dueño del puerto, y cualquier otro proceso que conoce de la existencia de dicho puerto puede usarlo.
Por otro lado, un puerto que es propiedad del sistema operativo es independiente, y no es unido a ningún proceso. El sistema operativo provee mecanismos que permite a los procesos:

- Crear un puerto.
- Enviar y recibir mensajes a través de un puerto.
- Destrozar un puerto.

El proceso que crea un puerto es el propietario del mismo por defecto, y es el único que puede recibir mensajes a través de éste buzón. Sin embargo, la posesión y el privilegio de recibir mensajes puede ser pasada a otro proceso a través de una llamada de sistema. Por supuesto, ésta propiedad podría derivar en múltiples recibidores de un mismo puerto. Los procesos también pueden compartir un puerto a través de la creación de procesos. Por ejemplo, si un proceso P crea el puerto A, y luego crea un nuevo proceso Q, P y Q comparten el buzón.

Buffering: un enlace tiene alguna capacidad que determina el número de mensajes que puede residir en él temporariamente. Esta propiedad puede ser vista como una cola de mensajes unidas al enlace. Básicamente existen tres formas en las cuales puede ser implementada tal cola:

- *Capacidad cero:* la cola tiene un largo máximo de 0; así, el enlace no tiene ningún mensaje esperando en él. En este caso, el emisor debe esperar hasta que el receptor reciba el mensaje. Los dos procesos deben estar sincronizados para que la transferencia de un mensaje tome lugar.
- *Capacidad limitada:* La cola tiene un largo finito de *n*; así, como mucho n mensajes pueden residir en la cola. En caso de que la cola no este llena en el momento que un mensaje es enviado, es ubicado en la cola (ya sea que el mensaje es copiado o se mantiene un puntero al mensaje), y el emisor puede continuar ejecutando sin la necesidad de esperar. Sin embargo, el enlace tiene una capacidad finita, por lo que si el enlace esta lleno, el emisor debe esperar hasta que halla espacio disponible en la cola.
- *Capacidad ilimitada:* La cola tiene un largo infinito; así, pueden esperar en el enlace cualquier cantidad de mensajes. El emisor nunca espera.

La capacidad cero se refiere a veces a un sistema de mensajes sin buffering; en cambio los otros dos proveen buffering automático.
Notemos que, en los casos donde la capacidad no es cero, el proceso no conoce si un mensaje ha llegado a o no a destino luego de que la operación **send** fue realizada. Si ésta información es crucial para la computación, el emisor debe comunicarse explícitamente con el receptor para averiguar si el mensaje fue recibido. Por ejemplo, supongamos que el proceso P envía un mensaje al proceso Q y puede continuar su ejecución solo si el mensaje enviado llego a destino. El proceso P ejecutara la secuencia:

send(Q, mensaje)
receive(Q, mensaje)

El proceso Q ejecutara:

receive(P, mensaje)
send(P, "acuse de recibo")

A tales procesos se dice que están comunicados asincrónicamente.
Hay casos especiales que no entran en ninguna de las categorías que se vieron:

- El proceso que envía el mensaje nunca es demorado (es decir, nunca se queda esperando). Sin embargo, si el proceso receptor del mensaje no recibió este primer mensaje antes de que el proceso emisor envíe otro mensaje, entonces el primer mensaje estará perdido. La ventaja de este esquema es que los mensajes grandes no necesitan ser copiados más de una vez. La mayor desventaja es que la tarea de programación es más difícil. Los procesos necesitan una sincronización explícita para estar seguros si el mensaje está o no perdido, y que el emisor y el receptor no manipulen simultáneamente el buffer.
- El proceso que envía un mensaje es retrasado hasta que recibe una respuesta. En este sistema, los mensajes son de un tamaño fijo (ocho palabras). Un proceso P que envía un mensaje es bloqueado hasta que el proceso receptor ha recibido el mensaje y envía una respuesta de ocho palabras por medio de la primitiva **reply**(P, mensaje). El mensaje de respuesta sobreescribe el buffer del mensaje original. La única diferencia entre las primitivas **send** y **reply** es que un **send** causa que el proceso que envía el mensaje sea bloqueado, mientras que **reply** hace que tanto el proceso emisor y el proceso receptor continúen sus ejecuciones inmediatamente.
  Este método de comunicación sincrónica se puede extender a un sistema RPC (call procedure remote), el cual esta basado en que una llamado a un procedimiento o subrutina en un sistema de único procesador actúa exactamente como un sistema de mensajes en el cual el emisor se bloquea hasta que reciba una respuesta. Este mensaje es entonces como el llamado a una subrutina, y el mensaje de retorno contiene el valor de la subrutina computada. De esta manera, el siguiente paso lógico es que los procesos concurrentes sean capaces de llamarse entre sí como subrutinas usando RPC.

Condiciones de excepción: Un sistema de mensajes es muy útil en un entorno distribuido, donde los procesos pueden residir en diferentes maquinas. En tales ambientes, la probabilidad de que ocurra un error durante la comunicación (y procesamiento) es mucho más grande que la que existe en un entorno de una única maquina. En una única maquina, los mensajes son usualmente implementados con el método de memoria compartida. En caso de que ocurra un fallo, entonces el sistema entero falla. En un entorno distribuido, sin embargo, los mensajes son transferidos por una línea de comunicación, y la falla de un sitio (o enlace) no necesariamente debe resultar en la falla del sistema completo.
Cuando una falla ocurre en un sistema centralizado o distribuido, toma lugar algunos sistemas de recuperación de errores (excepciones). Veremos brevemente algunas de las condiciones de excepción que el sistema puede manejar en el esquema de mensajes.

- ***El proceso termina:*** Un emisor o un receptor puede terminar antes de que un mensaje sea procesado. Esta situación provocara la existencia de mensajes que nunca serán recibidos, o procesos esperando por mensajes que nunca serán enviados. Consideremos dos casos:
    1. Un proceso receptor P puede esperar por un mensaje del proceso Q que ha terminado. Si no es tomada ninguna acción, P estará bloqueado por siempre. En este caso, puede optar por terminar también P o avisarle a P que Q ha terminado.
    2. El proceso P puede enviar un mensaje a Q que ha terminado. En el esquema de buffering automático no se produce daño: P simplemente continúa con su ejecución. En caso de que P necesite saber si el mensaje fue procesado por Q, éste debe explícitamente programar por un acuse de recibo. En el caso de que no halla buffering, P será bloqueado por siempre. Como en el caso 1, el sistema puede optar por terminar P o notificar a P que Q ha terminado.

- ***Perdida de mensajes:*** Un mensaje desde el proceso P al proceso Q se puede perder en algún lugar de la comunicación de red, falla del hardware, o falla en la línea de comunicación. Hay tres métodos básicos para tratar con este evento:
    1. El sistema operativo es el responsable de detectar éste evento y de retransmitir el mensaje.
    2. El proceso emisor es el responsable de detectar éste evento y de retransmitir el mensaje, en caso de que quiera hacerlo.
    3. El sistema operativo es el responsable de detectar éste evento; éste entonces notifica al proceso emisor que el mensaje se ha perdido. El proceso emisor puede elegir lo que le convenga.

  La pérdida de un mensaje se puede detectar por medio del uso de timeouts. Cuando un mensaje es enviado siempre retorna un mensaje de respuesta o un acuse de recibo. El sistema operativo o el proceso puede especificar un intervalo de tiempo en el cual éste espera que llegue el mensaje de acuse de recibo. En caso de que éste periodo de tiempo termine antes de que llegue el mensaje de acuse de recibo, el sistema operativo (o el proceso) puede asumir que el mensaje se perdió, y que el mensaje será reenviado. Puede ocurrir sin embargo que el mensaje no se perdió sino que el mensaje de acuse tardo más en llegar que el tiempo en el cual esta determinado el timeout. En este caso, se pueden obtener múltiples copias del mismo mensaje. Ante esto, debe existir un mecanismo para distinguir entre los diferentes tipos de mensajes.

Un ejemplo: Mach. Como ejemplo de un sistema operativo basado en pasaje de mensajes, consideremos el sistema operativo Mach. El kernel del Mach soporta la creación y destrucción de múltiples tareas, las cuales son similares a los procesos pero tiene múltiples threads de control. La mayor parte de la comunicación en Mach, incluyendo la mayoría de las llamadas la sistema y toda la información entre tareas, es llevada a cabo por mensajes. Los mensajes son enviados y recibidos por casillas llamadas puertos (en la nomenclatura Mach).
Las llamadas al sistema son hechas vía mensajes. Cuando se crea una tarea, también se crean dos puertos: el puerto del kernel y el puerto de Notificación. El puerto del kernel lo usa el kernel para comunicarse con la tarea. El kernel envía la notificación de los eventos que ocurren al puerto de Notificación. Solo se necesitan tres llamadas al sistema para la transferencia del mensaje. La llamada msg_send envía un mensaje al puerto. Un mensaje es recibido vía msg_receive. Los llamados a procedimientos remotos (RPC) son ejecutados vía msg_rpc, el cual envía un mensaje y espera por un mensaje de retorno del emisor.
La llamada al sistema port_allocate crea un nuevo puerto y asigna espacio para su cola de mensajes. El tamaño máximo de la cola de mensajes por defecto es de 8 mensajes. La tarea que crea el puerto es la dueña del puerto.

Al inicio, el puerto esta vacío de mensajes. A medida que los mensajes son enviados al puerto, estos se copian en él. Todos los mensajes tienen la misma prioridad. Mach garantiza que los múltiples mensajes enviados por un mismo emisor son encolados en una orden de tipo FIFO, pero no garantiza un orden absoluto. Por ejemplo, los mensajes enviados por diferentes emisores se pueden ordenar de cualquier manera.
Las operaciones del emisor y el receptor son bastante fáciles. Por ejemplo, cuando se envía un mensaje al puerto, el puerto puede estar lleno. Si el puerto no esta lleno, el mensaje se copia en el puerto y el thread emisor continúa. Si el puerto esta lleno, el thread emisor tiene cuatro opciones:

- Esperar indefinidamente hasta que haya lugar en el puerto.
- Esperar a lo sumo n milisegundos.
- No esperar, sino retornar inmediatamente.
- Guardar el mensaje temporariamente en cache. Un mensaje se le puede dar al sistema operativo ya que el puerto de destino esta lleno. Cuando el mensaje ya tiene lugar, se le envía un mensaje al emisor.

Un ejemplo: Windows NT. El sistema operativo Windows NT es un ejemplo de diseño moderno de un sistema que emplea la modularidad para incrementar la funcionalidad y disminuir el tiempo empleado para agregar nuevas características. NT provee soporte para diferentes sistemas operativos o subsistemas con programas de aplicación que se comunican con un mecanismo de paso de mensajes. Los programas de aplicación pueden ser considerados como clientes del servidor de NT.
La facultad de paso de mensajes en NT es llamada Local Procedure Call (LPC). El LPC en NT se usa para comunicar dos procesos que están en la misma maquina. NT, así como Mach, usa objetos puerto para establecer y mantener la comunicación entre dos procesos. Cada cliente que llama a un subsistema necesita un canal de comunicación el cual es proveído por un puerto y nunca es heredado. Existen dos tipos de puertos en NT: los puertos de conexión y los puertos de comunicación, el cual en realidad son los mismos pero difieren en su nombre de acuerdo el uso que se les da.

5. Scheduling de CPU

El scheduling de la CPU es lo básico de los sistemas operativos multiprogramados. Por medio del cambio de la CPU entre los diferentes procesos, el sistema operativo puede hacer que la computadora funcione de manera más eficiente.

Conceptos básicos

El objetivo de la multiprogramación es el de tener siempre un proceso corriendo, para maximizar la utilización de la CPU. En un sistema de único procesador, nunca habrá más que un proceso corriendo. En caso de que haya más de un proceso, el resto deberá esperar hasta que la CPU este libre y se les puede ser asignada.
La idea de la multiprogramación es simple. Un proceso es ejecutado hasta que debe esperar, típicamente, por la completitud de algún pedido de I/O. En una computadora simple, la CPU estaría ociosa. Todo éste tiempo que queda esperando es tiempo gastado, ya que no se logra trabajo útil. Con multiprogramación, se trata de usar éste tiempo productivamente. Varios procesos son mantenidos en memoria a la vez. Cuando un proceso tiene que esperar, el sistema operativo saca el proceso que estaba en la CPU y se la otorga a otro proceso.
El scheduling es una función fundamental de un sistema operativo. Casi todos los recursos de la computadora son scheduled antes de su uso. La CPU es, por supuesto, uno de los recursos principales de la computadora. Así, su scheduling es muy importante para el diseño del sistema operativo.

CPU scheduler: Cuando la CPU pasa a estar ociosa, el sistema operativo debe seleccionar uno de los procesos que esta en la cola de listos para ser ejecutados. Este proceso de selección es llevado a cabo por el scheduler de corto plazo (o scheduler de CPU). El scheduler selecciona uno de los procesos en memoria que están listos para ser ejecutados, y le asigna la CPU.
Note que la cola de listos no necesariamente debe ser una cola de tipo FIFO (first in-first out). Según el algoritmo de scheduling, la cola de listos puede ser una cola de tipo FIFO, una cola de prioridades, un árbol, o simplemente una lista enlazada desordenada. Los elementos de la cola son generalmente PCBs de procesos.

Scheduling desalojadores: Las decisiones del scheduler del CPU pueden tomar lugar bajo las siguientes cuatro circunstancias:

1. Cuando un proceso cambia desde el estado de corriendo al estado de esperando (por ejemplo, por un pedido de I/O, o la invocación de la llamada al sistema **wait** para que finalice uno de sus procesos hijos).
2. Cuando un proceso cambia desde el estado de corriendo al estado de listo (por ejemplo, cuando ocurre una interrupción).
3. Cuando un proceso cambia desde el estado de esperando al estado de listo (por ejemplo, por la completitud de la I/O).
4. Cuando un proceso termina.

Por las circunstancias 1 y 4, no existe elección en términos del scheduler. Un nuevo proceso (si es que existe uno en la cola de listos) se debe seleccionar para la ejecución. Hay sin embargo una elección para las circunstancias 2 y 3.
Cuando el scheduler toma lugar bajo las circunstancias 1 y 4, se dice que el esquema del scheduler es *nonpreemptive (sin desalojo)*; de otro modo, el esquema del scheduler es *preemptive (con desalojo)*. Bajo el scheduler sin desalojo, una vez que la CPU ha sido asignada a un proceso, éste mantiene la CPU hasta que la libera ya sea porque terminó, o porque cambio al estado de espera. Este método de scheduling es utilizado por el sistema operativo Windows 3.1 y Macintosh.

Los scheduler con desalojo requieren un costo. Consideremos el caso de dos procesos compartiendo datos. Un proceso puede estar en el medio de la modificación del dato cuando es desalojado y se le otorga la CPU al otro proceso. El segundo proceso puede tratar de leer el dato, el cual esta actualmente en un estado inconsistente. Ante esto, se necesitan mecanismos que coordinen los accesos a los datos compartidos.
La prevención también tiene un efecto en el diseño del kernel del sistema operativo. Durante el procesamiento de una llamada al sistema, el kernel puede estar ocupado con una actividad de un proceso. Tales actividades pueden involucrar un cambio importante de datos del kernel (por ejemplo, colas de I/O). ¿Qué ocurre si el proceso es desalojado en el medio de estos cambio, y el kernel (o el driver del dispositivo) necesita leer o modificar la misma estructura?. Se produce un caos. Algunos sistemas operativos, incluyendo la mayoría de las versiones de UNÍS, tratan este problema ya sea esperando que se complete la llamada al sistema, o que tome lugar un bloqueo de I/O antes de que tome lugar un context switch. Este esquema asegura que la estructura del kernel es simple, ya que el kernel no desaloja a un proceso mientras las estructuras de datos del kernel están en un estado inconsistente.

Dispatcher: Otro componente que involucra la función de scheduling de la CPU es el *dispatcher*. El dispatcher es un modulo que da el control de la CPU al proceso seleccionado por el scheduler de corto plazo. Sus funciones son:

- Switching context.
- Switching al modo usuario.
- Saltar al lugar apropiado en el programa del usuario para recomenzar el programa.

El dispatcher debe ser tan rápido como sea posible, ya que éste es invocado en cada cambio de proceso. El tiempo que le toma al dispatcher parar un proceso, realizar el cambio y comenzar a correr otro se llama *latencia del dispatcher*.

Criterios de scheduling

Los diferentes algoritmos de scheduling tienen diferentes propiedades y pueden favorecer a una u otra clase de procesos. La elección de cual algoritmo usar en una situación particular se debe considerar las propiedades de los diferentes algoritmos. Los criterios que son usados para comparar los diferentes algoritmos de scheduling son:

- *Utilización de la CPU:* lo que se desea es que la CPU esté tan ocupada como sea posible.
- *Throughput:* si la CPU esta ocupada ejecutando procesos, entonces el trabajo esta siendo realizado. Una medida de trabajo es el número de trabajos que están completos por unidad de tiempo, llamado Throughput. Para procesos largos, ésta medida puede ser un proceso por hora; para transacciones cortas llega a ser de 10 procesos por segundo.
- *Tiempo de turnaround:* desde el punto de vista de un proceso en particular, el criterio importante es cuanto tiempo le llevo a la CPU ejecutarlo. El intervalo de tiempo desde que el proceso es declarado como tal, hasta que la CPU lo completa se le llama tiempo turnaround. Este tiempo incluye la suma de los periodos gastados esperando para conseguir lugar en la memoria, esperando en la cola de listos, ejecutando en la CPU, y haciendo I/O.
- *Tiempo de espera:* los algoritmos de scheduling de la CPU no afectan la cantidad de tiempo durante el cual un proceso ejecuta o hace I/O; al scheduling solo afecta la cantidad de tiempo que el proceso gasta esperando en la cola de listos. El tiempo de espera es la suma de los periodos gastados esperando en la cola de listos.
- *Tiempo de respuesta:* En un sistema interactivo, el tiempo de turnaround puede no ser el mejor criterio. A menudo, un proceso puede producir una salida bastante rápido y puede continuar calculando nuevos resultados mientras los anteriores resultados están siendo mostrados al usuario. Así, otra medida es el tiempo desde que se realiza un pedido hasta que se produce la primer respuesta. Esta cantidad, llamada tiempo de respuesta, es la cantidad de tiempo que le

toma responder, pero no el tiempo que le toma presentar la salida. El tiempo de turnaround es generalmente limitado por la velocidad de los dispositivos de salida.

Es deseable que se maximice la utilización de la CPU y el throughput, y minimizar el tiempo de turnaround, el tiempo de espera, y el tiempo de respuesta. En la mayoría de los casos se optimiza la cantidad promedio. Sin embargo, hay circunstancias en el cual es deseable optimizar los valores mínimos o máximos, en lugar de la medida promedio. Por ejemplo, para garantizar que todos los usuarios tendrán un buen servicio, lo que se quiere es minimizar el máximo tiempo de respuesta.

Algoritmos de scheduling

El scheduling de la CPU trata con el problema de decidir cual de los procesos que están en la cola de listos será al próximo que se le asigne la CPU. Existen varios tipos de algoritmos.

Primero en entrar, primero en ser servido: El algoritmo más simple es el FIFO. Con este esquema, el proceso que pide la CPU primero es al que primero se le otorga. Su implementación puede ser fácilmente realizada con un lista FIFO. Cuando un proceso entra a la cola de listos, su PCB se une al final de la lista. Cuando la CPU esta libre, ésta se le asigna al proceso que esta en la cabeza de la lista.
El promedio de tiempo de espera bajo éste algoritmo es bastante grande.
Este algoritmo no es desalojador. Una vez que la CPU se le asigno a un proceso, el proceso mantiene la CPU hasta que la libera, ya sea porque finalizo o por un pedido de I/O.

El trabajo más corto es el primero en ser servido (Shortest-Job-First: SJF): Este algoritmo asocia a cada proceso un número el cual es el largo de tiempo que el proceso utilizara la CPU en la siguiente vez. Al estar disponible la CPU, ésta se le asigna al proceso que tiene el tiempo de uso de la CPU más corto. En caso de que dos procesos tengan el mismo tiempo, se utiliza el mecanismo FCFS entre ellos. Se debe tener en claro que éste número que se asocia a cada proceso es el tiempo que utilizará el proceso la CPU en el instante siguiente y no el tiempo total que el proceso necesita la CPU.
El algoritmo SJF es probablemente optimal en el sentido que retorna el menor promedio de tiempo de espera de los procesos para un conjunto de procesos dados. Moviendo un proceso más corto antes de un largo, el tiempo de espera del proceso corto decrece, decreciendo también el tiempo promedio de espera.
La mayor dificultad de este algoritmo es conocer cual es el largo del siguiente pedido de la CPU. Este algoritmo es usado generalmente en scheduling de largo plazo. Aunque este algoritmo sea optimal, no puede ser usado en scheduling corto ya que no existe forma de conocer cual será el tiempo de CPU que utilizara un proceso. Una idea es la de aproximar el scheduling de SJF, tratando de predecir el tiempo. La idea es suponer que el tiempo de pedido de la CPU será similar al tiempo pedido anteriormente. Este tiempo es predecido por medio de un promedio exponencial.

Scheduling de prioridades: En este algoritmo se asocia a cada proceso una prioridad y la CPU se le asigna al proceso que tenga la prioridad más alta (es decir, el número más pequeño). Los procesos con igual prioridad se les aplica FCFS.
Las prioridades se pueden definir externamente como internamente. Las prioridades internas utilizan algunas medidas para computar la prioridad del proceso, como pueden ser, tiempo limite, requerimientos de memoria, el número de archivos abiertos, el largo de tiempo del próximo pedido de I/O. Las prioridades externas se imponen por un sistema que es externo al sistema operativo, tal como la importancia del proceso, el rango del usuario del cual el proceso es dueño, etc.
El scheduling de prioridades puede ser desalojador o no desalojador. Cuando un proceso arriva a la cola de listos, su prioridad se compara con la prioridad del proceso que se está corriendo. En un algoritmo de scheduling de prioridad de tipo desalojador, el proceso recién llegado se adueñara de la CPU en caso de que su prioridad sea más alta que la del proceso que se estaba ejecutando. En un scheduling de prioridad de tipo no desalojador, simplemente se ingresara el nuevo proceso en la cabeza de la lista de listos.

El mayor problema que tiene éste algoritmo es el problema de inanición (starvation). Un proceso que esta en la cola de listos pero carece de la CPU puede ser considerado bloqueado, esperando por la CPU. Un algoritmo de scheduling de prioridades puede que deje un proceso de prioridad baja esperando indefinidamente en la cola de listos por la CPU. Este problema puede ser solucionado con un sistema llamado aging (envejecimiento). Este sistema va aumentando la prioridad de un proceso que hace rato que está en la cola de listos (por ejemplo, cada cierto tiempo se le aumenta la prioridad en 1 punto).

Scheduling Round-Robin: Este algoritmo es designado para sistemas de tiempo compartido. Es similar al FCFS pero se le agrega el desalojo. Se define una pequeña unidad de tiempo llamada tiempo de quantum, el cual va desde 10 a 100 milisegundos. La cola de listos es tratada como una cola circular. El scheduler de la CPU le va asignando la CPU a cada uno de los procesos de la cola de listos por un tiempo de hasta 1 quantum a cada proceso. Para implementar dicho algoritmo, debemos tener una cola de procesos del tipo FIFO. Los procesos que llegan son ingresados en la cola de la lista. La CPU toma el primer proceso de la cola de listos, setea un timer para interrumpir luego de un quantum, y dispatche el proceso.
Ante esto, pueden ocurrir dos cosas:

- Que el proceso pueda tener un pedido de la CPU menor al tiempo de quantum, por lo que en este caso el proceso por si mismo liberara la CPU y el scheduler procederá al siguiente proceso.
- De otra forma, si el tiempo de pedido de la CPU era mayor al tiempo del quantum, el timer provocara una interrupción al sistema operativo. Ante esto, se ejecutara un context switch, y el proceso será puesto otra vez en la cola de la lista de listos. El scheduler de la CPU seleccionara entonces el siguiente proceso.

Siempre se le asigna a todos los procesos sólo 1 quantum. El rendimiento de éste algoritmo depende en gran medida del tiempo del quantum. En el caso extremo, si el quantum tiende a infinito, entonces éste algoritmo tiende a tener valores de promedio iguales al algoritmo FCFS.
En software, debemos considerar el efecto que produce el context switch en el rendimiento del scheduling RR. Asumamos que tenemos solo un proceso de 10 unidades de tiempo. En caso de que el quantum sea de 12 unidades, el proceso finalizara en menos de 1 quantum, por lo que no existe overhead. En caso de que el quantum sea de 6 unidades, el proceso requiere de 2 quantum resultando en un context switch. En caso de que el tiempo del quantum sea de 1 unidad, entonces se necesitan 9 context switch, disminuyendo en gran medida la ejecución del proceso. Así, lo que se desea es un tiempo de quantum bastante grande con respecto al tiempo de context switch.

Scheduling de colas multiniveles: Otra clase de scheduling se puede aplicar cuando los procesos se pueden dividir en grupos. Por ejemplo, una división común es hecha entre procesos foreground (interactivos), y procesos background (batch). Lo que diferencia a estos dos grupos es su tiempo de respuesta. Además, los procesos interactivos deben tener una prioridad definida externamente sobre los procesos de tipo batch.
Un algoritmo de scheduling de colas de multinivel lo que hace es dividir la cola de listos en varias colas separadas, como lo muestra la figura 5.6. A cada proceso que llega se le asigna algunas de las listas, generalmente basado en alguna propiedad del proceso tal como el tamaño de memoria, su prioridad, tipo de proceso, etc. Cada cola tiene su propio algoritmo de scheduling. Además, debe existir un scheduling entre las diferentes colas, el cual es generalmente implementado como un algoritmo fijo de elección. Por ejemplo, los procesos del sistema tienen prioridad absoluta sobre todos los demás tipos de procesos, etc. Cada cola tiene prioridad absoluta sobre las colas de prioridad más baja. Por ejemplo, ningún proceso en la cola batch podría ser ejecutado a menos que las colas de los procesos del sistema, los procesos interactivos y los procesos editores interactivos estén vacías. Ante esto, el algoritmo es desalojador, es decir, si supongamos que la cola de procesos del sistema esta vacía y esta ejecutando un proceso interactivo, y en ese momento llega a la cola de procesos del sistema un proceso, entonces el proceso interactivo que estaba ejecutando debe abandonar la CPU y la misma debe ser asignada al proceso del sistema.

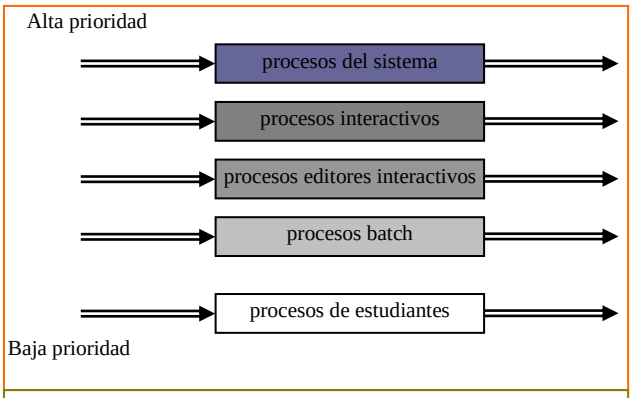


Figura 5.6 Scheduling de cola multinivel.

Otra posibilidad de algoritmo de scheduling entre las diferentes colas es asignarle a cada cola una cantidad de tiempo, por ejemplo, a la cola de procesos del sistema se le asignan 30 milisegundos, a la de procesos interactivos 15, etc.

Scheduling de regeneración de colas multinivel: Normalmente, en un algoritmo de scheduling de multinivel de colas, los procesos están asignados de entrada a un tipo de cola. Los procesos no se pueden mover entre las diferentes colas. Sin embargo, con el scheduling de regeneración de colas multinivel, un proceso se puede mover entre las diferentes colas. La idea es separar los procesos según la característica del siguiente pedido de la CPU. En caso de que un proceso utilice mucho tiempo de CPU, entonces será movido a una cola de baja prioridad. Este esquema pone limites a la I/O y pone a los procesos interactivos en la cola de mayor prioridad. De la misma forma, un proceso que estuvo esperando mucho tiempo en una cola de prioridad baja será movido a una cola de mayor prioridad, con lo que se evita el problema de inanición.

Scheduling para múltiple procesadores

Cuando están disponibles muchos procesadores, el problema de scheduling es más complejo. Como se vio anteriormente, no existe el algoritmo óptimo. A continuación se verán algunos de los problemas correspondientes al scheduling de múltiple procesadores. Tomaremos que todos los procesadores disponibles son idénticos, y que un procesador puede ejecutar cualquiera de los procesos que se encuentran en la cola de listos.

Aun en sistemas con procesadores homogéneos, hay casos donde el sistema tiene limitaciones. Consideremos un sistema con un dispositivo de I/O unido a un bus privado de un procesador. Los procesos que están deseando utilizar dicho dispositivo deben ser elegidos para correr en ese procesador, ya que de otra forma el dispositivo no estaría disponible.

En caso de que haya varios procesadores idénticos disponibles, entonces lo que se podría tener es una cola de listos para cada procesador. Pero podría pasar que un procesador esté ocioso ya que su cola de listos esta vacía, mientras existe otro procesador que esta sobrecargado. Para evitar esto, lo que se hace es tener una única cola de listos, donde en caso de existir un procesador disponible, se le asigna al siguiente proceso.

Bajo este último esquema, existen dos posibilidades de scheduling. La primera es que cada procesador implemente el algoritmo de scheduling. Ante esto, cada procesador recorrerá la cola de listos común y seleccionará el proceso a ejecutar. Si tenemos múltiples procesadores tratando de acceder y cambiar una estructura de datos común, se debe asegurar que dos procesadores no elijan el mismo proceso, y que no se pierdan procesos en la cola. La segunda posibilidad evita este problema fijando a un procesador la tarea de scheduling para los otros procesadores, creando así una estructura maestro-esclavo.

Scheduling para sistemas de tiempo real

En esta sección se verán los algoritmos de scheduling necesarios para poder soportar la computación en tiempo real.

La computación en tiempo real se divide en dos tipos. Los sistemas de tiempo real duros son requeridos para completar una tarea critica en una cantidad garantizada de tiempo. Generalmente, un proceso es declarado con la cantidad de tiempo limite que debe ser completado o realizar un I/O. Ante esto, el scheduler puede aceptar el proceso, garantizando que será completado en el tiempo establecido, o lo rechaza. Para que el scheduler garantice que completara un proceso, necesita saber que cantidad de tiempo lleva cada función del sistema operativo. En sistemas con almacenamiento secundario o memoria virtual, ésta garantía es imposible que se dé, ya que estos subsistemas causan una inevitable e imprevisible variación en la cantidad de tiempo que tardan para ejecutar un proceso. Así, los sistemas de tiempo real duros están compuestos de software de propósito especial corriendo en un hardware dedicado a sus procesos críticos, y carece de la funcionalidad completa de las modernas computadoras y sistemas operativos.

Los sistemas de tiempo real suaves o blandos son menos restrictos. Su implementación necesita cuidado en el diseño de los schedulers y los aspectos relacionados con el sistema operativo. Primero, el sistema debe tener scheduler de prioridades, y los procesos de tiempo real deben tener la prioridad más alta. La prioridad de los procesos de tiempo real no debe disminuir en el tiempo, aun aunque la prioridad de los procesos de tiempo no real puedan hacerlo. Segundo, la latencia de dispatch debe ser baja.

Es difícil asegurar la propiedad de que la latencia del dispatch debe ser baja. El problema es que muchos sistemas operativos, incluyendo muchas versiones de UNIX, están forzadas a esperar por la completitud de una llamada al sistema, o por un bloque de I/O que toma lugar antes de hacer el context switch. La latencia del dispatch en tales sistemas puede ser larga, ya que algunas llamadas al sistema son complejas y algunos dispositivos de I/O son lentos.

Para mantener la latencia del dispatch baja, se necesita permitir que las llamadas al sistema puedan ser desalojadas. Existen varias formas de conseguir este objetivo. Una es la de insertar puntos de desalojo en las llamadas al sistema largas, el cual chequea si un proceso de alta prioridad debe ser ejecutado. En tal caso, es decir, que exista un proceso de alta prioridad, toma lugar un context switch y, cuando el proceso de alta prioridad termina, el proceso interrumpido continúa con la llamada al sistema. Los puntos de desalojos se deben ubicar en lugares seguros del kernel, es decir, cuando las estructuras de datos del kernel no están siendo modificadas.

Otra forma de tratar con el desalojo es el de hacer todo el kernel desalojable. Ante el hecho de que la actual operación esta asegurada, todas las estructuras de datos del kernel deben estar protegidas por medio del uso de varios mecanismos de sincronización. Con éste método, el kernel siempre puede ser desalojable, ya que cualquier dato del kernel que esta siendo modificado esta protegido de modificaciones que puedan hacer los procesos de alta prioridad. Este método es el utilizado por Solaris 2.

En caso de que un proceso de alta prioridad necesite modificar un dato del kernel que en este momento esta siendo modificado por un proceso de baja prioridad, entonces el proceso de alta prioridad debe esperar hasta que el proceso de baja prioridad termine.

La figura 5.8 muestra la latencia del dispatch.

La fase de conflicto de la latencia del dispatch tiene tres componentes:

- Desalojo de cualquier proceso corriendo en el kernel.
- Proceso de baja prioridad liberando los recursos necesitados por el proceso de alta prioridad.
- Context switching del actual proceso al proceso de alta prioridad.

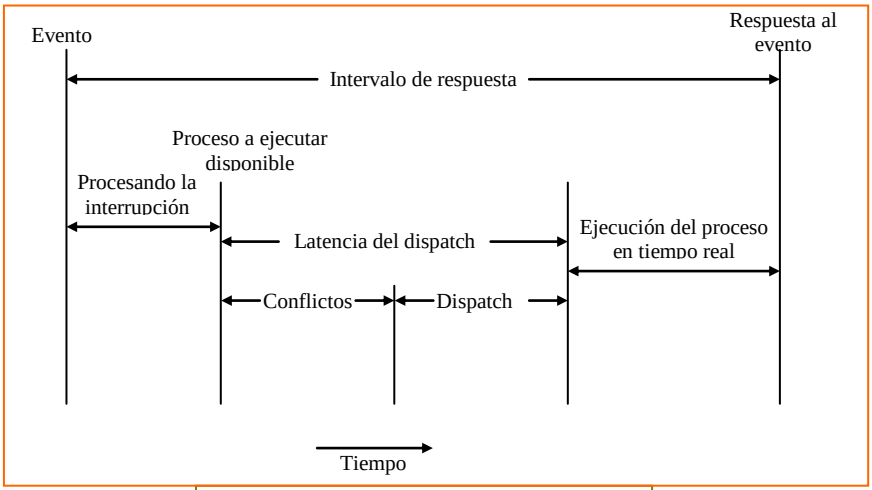


Figura 5.8 Latencia del dispatch.

6. Sincronización de procesos

Un proceso cooperador es aquel que afecta o es afectado por otros procesos que están ejecutando en el sistema. Los procesos cooperadores pueden o compartir directamente un espacio de direcciones lógicas (ambos: código y datos), o pueden compartir datos sólo a través de archivos. El caso anterior es conseguido por medio del uso de procesos livianos o threads. El acceso concurrente a los datos compartidos puede resultar en la inconsistencia de éstos datos. En ésta parte se verán mecanismos que aseguren la ejecución ordenada de los procesos cooperadores que comparten espacio de direcciones lógicas para que los datos estén consistentes.

Conceptos básicos

En el capítulo 4 se vio un modelo de un sistema consistente de un número de procesos cooperando secuencialmente, todos corriendo asincrónicamente y posiblemente compartiendo datos. Este modelo fue ilustrado con el esquema de buffer limitado.

Retornemos a la solución de la memoria compartida para el problema del buffer limitado que se vio en la sección 4.4. Como se señaló, nuestra solución permitía que como mucho n-1 ítems estén en el buffer en un mismo tiempo. Supongamos que queremos modificar el algoritmo para remediar ésta deficiencia. Una posibilidad es la de agregar una variable entera *counter*, inicializada en 0, el cual es decrementada cada vez que se elimina un ítem del buffer, y es incrementada cada vez que se agrega un ítem en el buffer. El código del proceso productor puede ser modificado como sigue:

```
repeat
        ...
        producir un ítem en nextp
        ...
        while counter = n do no-op;
        buffer[in] := nextp;
        in := in + 1 mod n;
        counter = counter + 1;
until false;
```

El código del proceso consumidor seria:

```
repeat
        while counter = 0 do no-op;
        nextc = buffer[out];
        out := out + 1 mod n;
        counter = counter - 1;
        ...
        consumir el ítem en nextc
        ...
until false;
```

Aunque ambas rutinas parecen estar correctamente escritas, no funcionaran bien en el momento de ser ejecutadas concurrentemente. Como ejemplo, supongamos que el valor de la variable *counter* es 5, y que tanto el proceso productor como el consumidor están a punto de ejecutar la sentencia "*counter* = *counter* + 1" y "*counter* = *counter* – 1" respectivamente, concurrentemente. Siguiendo con la ejecución de estas dos sentencias, el valor de la variable counter puede ser 4, 5 o 6; y el único resultado correcto es 5, el cual es generado correctamente en caso de que el productor y el consumidor ejecuten separadamente.

Veamos porque el valor de counter puede ser incorrecto. Notemos que la sentencia "counter = counter + 1" puede ser implementada en lenguaje de maquina como sigue:

```
register1 := counter;
register1 := register1 + 1;
```

counter = $register_1$;

donde $register_1$ es un registro local de la CPU. Similarmente, la sentencia "counter = counter – 1" puede ser implementada como sigue:

$register_2$:= counter;
$register_2$:= $register_2$ - 1;
counter = $register_2$;

donde nuevamente $register_2$ es un registro local de la CPU. Note que los registros 1 y 2 podrían ser los mismos, ya que los contenidos de los registros se almacenan y luego se vuelven a cargar por el manejador de las interrupciones.
La ejecución concurrente de la sentencia "counter = counter + 1" y "counter = counter – 1" es equivalente a una ejecución secuencial donde las instrucciones de maquinas anteriores podrían entremezclarse. Una de éstas mezcladas podría ser:

T_0:	productor	ejecuta	$register_1$:= counter	{$register_1$ = 5}
T_1:	productor	ejecuta	$register_1$:= $register_1$ + 1	{$register_1$ = 6}
T_2:	consumidor	ejecuta	$register_2$:= counter	{$register_2$ = 5}
T_3:	consumidor	ejecuta	$register_2$:= $register_2$ - 1	{$register_2$ = 4}
T_4:	productor	ejecuta	counter := $register_1$	{counter = 6}
T_5:	consumidor	ejecuta	counter := $register_2$	{counter = 4}

Note que llegamos a un estado incorrecto de counter, el cual recuerda que hay 4 buffers llenos, cuando en realidad hay 5. En caso de que cambiemos de lugar las sentencias T_4 y T_5, llegaremos a un valor de 6 para counter. Este error se debe a que permitimos a los procesos manipular la variable counter concurrentemente. Para evitar que este error se produzca debemos asegurar que sólo un proceso a la vez podrá manipular la variable counter. Ante esto se necesita alguna forma de sincronización de procesos. En este capítulo se verán métodos que garantizan la consistencia de los datos compartidos.

Problema de la sección crítica

Consideremos un sistema consistente de n procesos {P_0, P_1, ..., P_{n-1}}. Cada proceso tiene un segmento de código, llamada la sección critica, en el cual el proceso puede estar cambiando variables comunes, modificando una tabla, escribiendo un archivo, etc. La característica importante del sistema es que cuando un proceso esta ejecutando la sección critica, a ningún otro proceso se le permitirá ejecutar en esta sección. Así, la ejecución de las secciones criticas para los procesos es mutuamente exclusiva. En éste protocolo, cada proceso debe pedir permiso para entrar a su sección critica. La parte de código que implementa este pedido es la sección de entrada (*entry*). La sección critica puede ser seguida por la sección de salida (*exit*). El resto del código es la sección restante (*remainder*).
Una solución al problema de la sección critica debe satisfacer los siguientes tres requerimientos:

1. *Exclusión mutua:* Si un proceso P_i esta ejecutando en su sección critica, entonces ninguno de los otros procesos pueden estar ejecutando en sus secciones criticas.
2. *Progreso:* Si no hay ningún proceso ejecutando en la sección critica y existen algunos procesos que desea entrar a sus secciones criticas, entonces solo aquellos procesos que no están ejecutando en su secciones de resto pueden participar en la decisión de cual proceso será el siguiente que entre a su sección critica, y ésta selección no puede ser pospuesta indefinidamente.
3. *Espera limitada:* Existe un número limite de veces que a otro procesos se les permite entrar a sus secciones criticas después de que un proceso ha hecho un pedido por entrar a su sección critica y antes de que el pedido es concedido.

Se asume que cada proceso no esta ejecutando a una velocidad de 0. Las soluciones de éste problema serán vistas más adelante. La soluciones no dependen de ninguna característica concerniente a las instrucciones de hardware o al número de procesadores que soporta el hardware. Igualmente, asumimos que las instrucciones en lenguaje de maquina básicas (load, store, test) son ejecutadas automáticamente. Esto es, si dos de tales instrucciones son ejecutadas concurrentemente, el resultado es equivalente si se ejecutan en cualquier orden. Así, si un load o un store son ejecutadas concurrentemente, el load obtendrá o el valor nuevo o el valor viejo pero no alguna combinación de los dos.
En el momento de presentar un algoritmo, definimos solo las variables usadas para el propósito de sincronización, y describe solo un proceso típico P_i el cual tiene la estructura mostrada en la figura 6.1. Las secciones entry y exit son cerradas en cajas para aumentar la importancia de éstos segmentos de código.

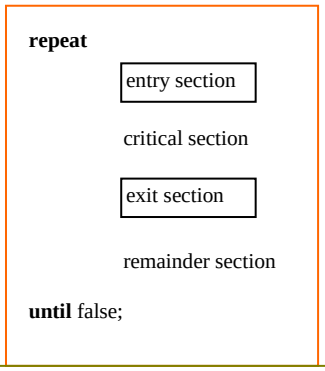


Figura 6.1 Estructura general de un proceso típico P_i.

Solución con dos procesos: En esta parte se aplicara la solución al problema anterior basándonos en la existencia de solo dos procesos P_0 y P_1. Por conveniencia no se presenta P_0 y P_1 sino P_i y P_j.

- *Primer algoritmo:* nuestra primera solución es la de utilizar una variable compartida *turn* inicializada en 0 (o en 1). Si turn = i, entonces al proceso P_i se le permite ejecutar en su sección critica. Esta solución se muestra en la figura 6.2. Esta solución asegura que solo un proceso a la vez puede estar en su sección critica. Sin embargo, no satisface el requerimiento de progreso, ya que ésta solución requiere una alternación estricta de los procesos en la ejecución de la sección critica. Por ejemplo, si turn = 0 y P_1 esta listo para entrar a la sección critica, P_1 no puede hacerlo, aunque P_0 este en su sección de resto.

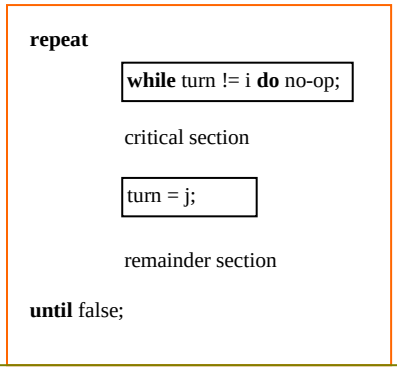


Figura 6.2 Estructura del proceso P_i en el algoritmo 1.

- *Segundo algoritmo:* El problema de la solución anterior es que no guarda suficiente información sobre el estado de cada proceso; solo retenía cual era el próximo proceso que podía entrar en su sección critica. Para remediar el problema se reemplazara la variable turn por el siguiente arreglo:

```
var flag: array[0..1] of boolean;
```

Los elementos del arreglo están inicializados a false. Si flag[i] es true, este valor indica que P_i esta listo para entrar en su sección critica. La estructura del proceso P_i se ve en la figura 6.3.

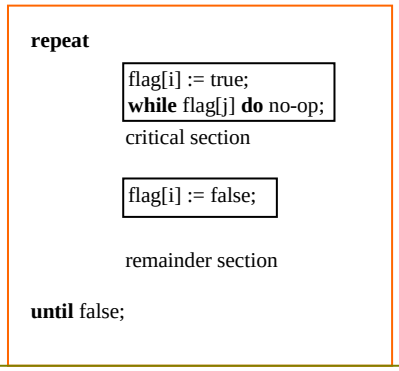


Figura 6.3 Estructura del proceso P_i en el algoritmo 2.

En este algoritmo, el proceso P_i primero setea flag[i] para que sea true, señalando que esta listo para entrar a su sección critica. Luego, P_i chequea para verificar que el proceso P_j no esté también listo para entrar a su sección critica. Si P_j estaba listo, entonces P_i esperará hasta que P_j haya indicado que salió de la sección (es decir, hasta que flag[j] sea false). En este punto, P_i entrará en la sección critica. Al salir de la sección, P_i seteara su bandera en false, permitiendo que el otro proceso (si es que estaba esperando) entre a su sección critica.
En ésta solución, el requerimiento de exclusión mutua se satisface, pero no ocurre lo mismo con el requerimiento de progreso. Para ver el problema, consideremos la siguiente secuencia de ejecución:

T_0: P_0 setea flag[0] = true
T_1: P_1 setea flag[1] = true

Ahora P_0 y P_1 quedaran en un loop por siempre en sus respectivas sentencias de while.
Este algoritmo es muy dependiente del momento exacto en que se encuentran ambos procesos. La secuencia podría haber sido derivada en un entorno donde hay varios procesadores ejecutando concurrentemente, o donde una interrupción (tal como una interrupción de timer) ocurre inmediatamente después de que T_0 es ejecutada, y la CPU es cambiando desde un proceso a otro.
Note que el cambio del orden de las instrucciones para setear flag[i] y testear el valor de flag[j] no resolverá el problema. En lugar de esto, llegamos a la situación donde ambos procesos podrían estar en la sección critica a la vez, violando el requerimiento de exclusión mutua.

- *Tercer algoritmo:* Por medio de la combinación de las ideas del primer y segundo algoritmo podemos obtener una solución correcta donde se cumplen los tres requerimientos. Los procesos comparten dos variables:

```
var flag: array[0..1] of boolean;
    turn: 0..1;
```

Inicialmente flag[0] = flag[1] = false, y el valor de turn no es importante (pero es 0 o 1). La estructura del proceso P_i se ve en la figura 6.4.
Para entrar a la sección critica, el proceso P_i primero setea flag[i] a true, y luego afirma que es el otro proceso el que desea entrar (turn = j). En el caso de que ambos procesos trataran de entrar al mismo tiempo, turn se estearía tanto a i como a j, pero solo una de estas asignaciones será la última, la otra asignación será sobrescrita por la que es última. El valor eventual de turn decidirá cual de los dos procesos será el que entre en la sección critica primero.

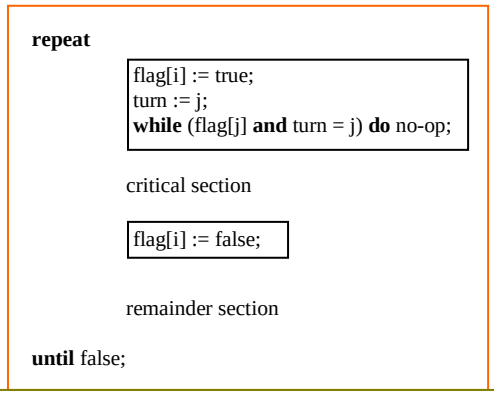


Figura 6.4 Estructura del proceso P_i en el algoritmo 3.

Veamos que tanto el requerimiento de exclusión mutua, el de progreso, y el de tiempo de espera limitado se cumplen. Para probar el primer requerimiento, notemos que cada entrada de P_i a su sección critica se produce solo si flag[j] = false y turn = i. Notemos también que en caso de que los dos procesos estén ejecutando a la ves en la sección critica, entonces tendría que pasar que flag[0] = flag[1] = true. Pero esto implica que P_0 y P_1 podrían no estar ejecutando con éxito sus sentencias **while**, ya que el valor de turn puede ser o 0 o 1, pero no ambos. Por lo tanto, uno de los procesos (digamos P_j) debe estar ejecutando con éxito la sentencia **while**, mientras que P_i tiene que ejecutar al menos una sentencia adicional ("turn = j"). Sin embargo, ya que al mismo tiempo flag[j] = true y turn = i, y ésta condición seguirá mientras P_j este en su sección critica, el resultado es el siguiente: se preserva la exclusión mutua.
Para proveer los requerimientos 2 y 3, notemos que un proceso P_i puede ser prevenido de entrar a la sección critica sólo si se queda atascado en el **while** con la condición flag[j] = true y turn = j. En caso de que P_j no este listo para entrar a la sección crítica, entonces flag[j] = false, por lo que P_i puede entrar a su sección critica. Si P_j tiene seteada flag[j] = true y también esta ejecutando en la sentencia **while**, entonces puede ser que turn = i o turn = j. Si turn = i, entonces P_i entrara a la sección critica. Si turn = j, entonces P_j entrara en la sección critica. Sin embargo, una vez que P_j sale de su sección critica, éste reseteará flag[j] a false, permitiendo que P_i entre ahora en la sección. Si P_j resetea flag[j] a verdadero, éste debe también setear turn en i. Así, ya que P_i no cambia el valor de la variable turn mientras esta ejecutando la sentencia **while**, P_i entrara en la sección critica (progreso) después de a lo sumo una entrada de P_j (espera limitada).

Soluciones para múltiple procesos: En esta sección se mostrara un algoritmo que resuelve el problema de la sección critica para n procesos. En un negocio cada cliente recibe un número y el cliente con el número más chico es el siguiente que recibe atención. Desafortunadamente, el algoritmo que veremos (algoritmo de bakery) no garantiza que dos procesos (clientes) no reciban el mismo número. En caso de empate, el proceso que tenga menor nombre es el que se sirve primero, esto es, si P_i y P_j reciben el mismo número y i < j, entonces P_i es servido primero. Ya que los nombres de los procesos son únicos y totalmente ordenados, nuestro algoritmo es completamente determinístico.
Las estructuras de datos comunes son:

```
var choosing: array[0..n-1] of boolean;
    number: array[0..n-1] of integer;
```

Inicialmente, las estructuras son inicializadas en false y 0. Por conveniencia definimos la siguiente notación:

- (a, b) < (c, d) si a < c o en caso de que a = c, si b < d.
- $\max(a_0, \ldots, a_{n-1})$ es un número, k, tal que k>= a_i para i = 0, ..., n-1.

La estructura del proceso P_i para el algoritmo bakery se ve en la figura 6.5.

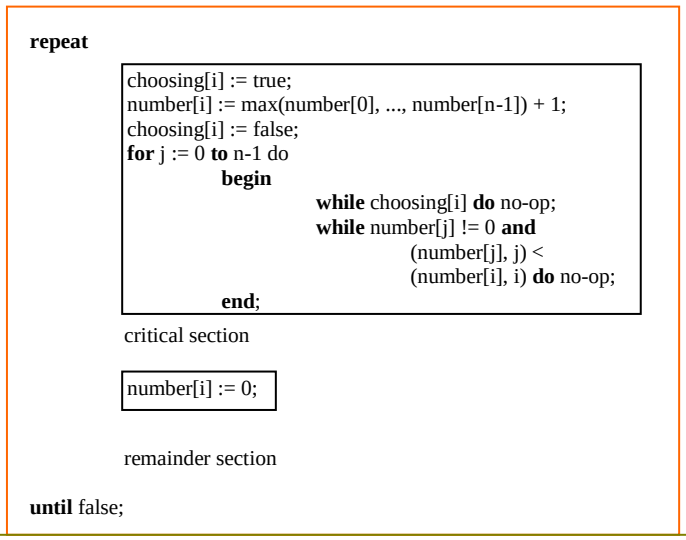


Figura 6.5 Estructura del proceso P_i en el algoritmo de bakery.

Para ver que este algoritmo es correcto, se necesita primero mostrar que si P_i esta en su sección critica y P_k (k != i) ya ha elegido su number[k] != 0, entonces (number[i], i) < (number[k], k).
Para probar la exclusión mutua, consideremos que P_i esta en su sección critica y P_k esta tratando de entrar a la sección critica de P_k. Cuando el proceso P_k ejecuta el segundo **while** para j = i, éste encuentra que:

- number[i] != 0
- (number[i], i) < (number[k], k).

Así, el loop continúa hasta que P_i deja su sección critica.
Para probar que los requerimientos de progreso y de espera limitada se cumplen, es suficiente observar que los procesos entran a sus secciones criticas sobre el FCFS básico.

Hardware de sincronización

En esta sección se verán algunas instrucciones de hardware que ayudan a resolver el problema de la sección critica.
El problema de la sección critica se podría resolver fácilmente en un uniprocesador si anulamos las interrupciones mientras una variable compartida esta siendo modificada. De esta manera, la secuencia de instrucciones se ejecutaran en el orden correcto sin interrupción.
Desafortunadamente, ésta solución no es posible en un sistema multiprocesador. Desactivar las interrupciones en un multiprocesador puede consumir mucho tiempo, por el pasaje del mensaje a todos los procesadores de que las interrupciones están momentáneamente deshabilitadas. Este paso del mensaje retarda la entrada en cada sección critica, y el sistema decrece en eficiencia. También, consideremos lo que ocurriría en el reloj del sistema, en caso de que el reloj sea cambiado por medio de interrupciones.
De esta manera, muchas maquinas proveen instrucciones de hardware especiales que nos permiten o testear o modificar el contenido de una palabra, o intercambiar el contenido de dos palabras, atómicamente.
La instrucción Test-and-Set se puede definir como lo muestra la figura 6.6. La característica más importante es que dicha función es ejecutada atómicamente, es decir, es una unidad in-interrumpible. Así, si dos instrucciones Test-and-Set son ejecutadas simultáneamente (cada una en una CPU diferente), serán ejecutadas secuencialmente en un orden especifico.

```
function Test-and-Set (var target:boolean) :bololean;
        begin
                Test-and-Set := target;
                target := true;
        end;
```

Figura 6.6 Definición de la instrucción Test-and-Set.

En caso de que la máquina soporte la instrucción Test-and-Set, entonces puede implementar la exclusión mutua por medio de la declaración de una variable boolean *lock*, inicializada en false. La estructura del proceso P_i se ve en la figura 6.7.

```
repeat
        while Test-and-Set(lock) do no-op;

        critical section

        lock = false;

        remainder section

until false;
```

Figura 6.7 Implementación de la exclusión mutua implementada con Test-and-Set.

La instrucción *Swap* opera con el contenido de dos palabras y como la anterior instrucción, también se ejecuta atómicamente (Figura 6.8).

```
procedure Swap (var a, b:boolean);
        var temp : boolean;
        begin
                temp := a;
                a := b;
                b := temp;
        end;
```

Figura 6.8 Definición de la instrucción Swap.

En caso de que la máquina soporte dicha instrucción, la exclusión mutua puede ser proveída como sigue. Se declara una variable boolean *lock* el cual se inicializa en false. Además, cada proceso tiene una variable local boolean llamada key. La estructura del proceso P_i y el procedimiento Swap se muestran en la figura 6.9.

```
repeat
        key := true;
        repeat
                Swap(lock, key);
        until key = false;

        critical section

        lock = false;

        remainder section

until false;
```

Figura 6.9 Implementación de la exclusión mutua con la instrucción Swap.

Estos algoritmos no satisfacen el requerimiento de tiempo limitado. En la figura 6.10 se muestra un algoritmo que sí satisface dicho requerimiento y los dos restantes. La estructuras de datos comunes son:

```
var waiting: array[0..n-1] of boolean
    lock: boolean;
```

Estas estructuras se inicializan en false.

```
var j: 0..n-1;
    key: boolean;
repeat
        waiting[i] := true;
        key := true;
        while waiting[i] and key do key := Test-and-Set(lock);
        waiting[i] := false;

        critical section

        j := i + 1 mod n;
        while (j != i) and (not waiting[j]) do j := j + 1 mod n;
        if j = i
            then lock := false
            else waiting[j] := false;

        remainder section

until false;
```

Figura 6.10 Exclusión mutua con espera limitada implementada con Test-and-Set.

Para ver que se cumple la exclusión mutua, veamos que el proceso P_i puede entrar a su sección critica solo si se cumple waiting[i] = false o key = false. Key es falsa solo si se ejecuta Test-and-Set. El primer proceso que ejecute Test-and-Set encontrara que key = false, por lo que todos los demás deben esperar. La variable waiting[i] es falsa solo si otro proceso deja su sección critica; solo un waiting[i] es falso, por lo que se mantiene la exclusión mutua.
Para proveer el requerimiento del progreso, notemos que los argumentos presentados por la exclusión mutua también se pueden aplicar aquí, ya que un proceso que sale de la sección critica o bien setea lock en falso o seteando waiting[j] a falso. Ambos permiten a un proceso que esta esperando para entrar a su sección critica que sea procesado.
Para proveer el limite de espera, notemos que cuando un proceso deja su sección critica, éste escanea el arreglo waiting en el ciclo (i + 1, i+ 2, ..., n – 1, 0, 1, ..., i – 1). Esto designa el primer proceso que esta en la sección de entrada (waiting[j] = true) como el siguiente a entrar en la sección critica.

Semáforos

Las soluciones anteriores son difíciles de aplicar en problemas más complejos. En ésta parte presentamos otra herramienta llamada *semáforo*, el cual no es más que una variable entera, donde es inicializada y accedida solo por dos operaciones atómicas: *wait* y *signal*. La definición de wait y signal es (S es un semáforo):

```
wait(S): while S<= 0 do no-op;
         S := S – 1;

signal(S): S := S + 1;
```

Las modificaciones que se le hagan al semáforo a través de las operaciones wait y signal deben ser realizadas en forma indivisible, es decir, en caso de que un proceso esté modificando el valor de un semáforo, ningún otro proceso puede estar simultáneamente modificando el valor del mismo semáforo. Además, en el caso de wait(S), el testeo del valor de S (S <= 0), y su posible modificación (S := S – 1) debe también ser ejecutada sin interrupción.

Uso: Podemos utilizar semáforos para tratar con el problema de la sección critica de los n procesos. Los n procesos comparten un semáforo, mutex (entendiendo por mutua exclusión) inicializado en 1. Cada proceso es organizado como se presenta en la figura 6.11.

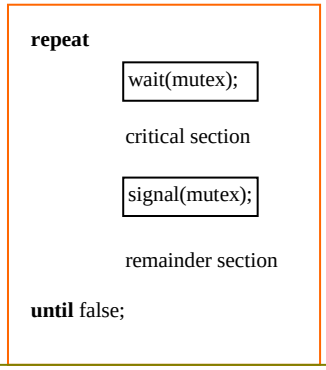


Figura 6.11 Implementación de la exclusión mutua con semáforos.

También podemos utilizar semáforos para resolver varios problemas de sincronización. Por ejemplo, consideremos dos procesos corriendo concurrentemente: P_1 con una sentencia S_1, y P_2 con una sentencia S_2. Supongamos que requerimos que S_2 sea ejecutada solo después que S_1 fue completada. Podríamos resolver este problema haciendo que P_1 y P_2 compartan un semáforo synch, inicializado en 0, e insertando las sentencias:

S_1;
signal(synch);

en el proceso P_1, y las sentencias:

wait(synch);
S_2;

en el proceso P_2. Ya que synch esta inicializado en 0, P_2 ejecutará S_2 solo después de que P_1 haya invocado signal(synch), el cual esta después de S_1.

Implementación: La mayor desventaja de la exclusión mutua que se presento hasta ahora es que todas las soluciones requieren espera ocupada. Mientras un proceso está en la sección critica, cualquier otro proceso que trate de entrar en su sección critica debe hacer un loop continuo en el código de entrada. Este loop continuo es un claro problema en los sistemas multiprogramados, donde una única CPU es compartido a través de varios procesos. La espera ocupada gasta ciclos de CPU que algún otro proceso puede ser capaz de utilizar productivamente. Este tipo de semáforos es llamado spinlock (ya que el proceso da vueltas (spin), mientras espera por el bloqueo (lock)). Spinlock son muy útiles en sistemas multiprocesadores. La ventaja de un spinlock es que no requiere context switch cuando un proceso debe esperar en un bloqueo (ya que no abandona la CPU), y un context switch puede llevar un considerable tiempo. Así, cuando las trabas son mantenidas por cortos tiempos, spinlocks son muy útiles.
Para superar la necesidad de la espera ocupada podemos modificar la definición de wait y signal. Cuando un proceso ejecuta wait, y encuentra que el valor del semáforo no es positivo, éste debe esperar. Sin

embargo, en lugar de hacer una espera ocupada, el proceso puede bloquearse a sí mismo. La operación de bloqueo ubica al proceso en la cola de espera asociada con el semáforo, y el estado del proceso es cambiado a esperando. Luego, el control es transferido al scheduler de la CPU, el cual selecciona otro proceso para ejecutar.
Un proceso que esta bloqueado, esperando por el semáforo S, puede ser reiniciado cuando algún otro proceso ejecuta la operación de signal. El proceso es reiniciado por medio de la operación wakeup, el cual cambia al proceso desde el estado de esperando al estado de listo. El proceso es entonces ubicado en la cola de listos. La CPU puede o no cambiar desde el proceso que estaba corriendo al nuevo proceso que fue despertado, dependiendo del algoritmo de scheduling de CPU.
Para implementar los semáforos bajo ésta definición, definimos un semáforo como un record:

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

Cada semáforo tiene un entero y una lista de procesos. Cuando un proceso debe esperar en un semáforo, éste es agregado a la lista de procesos. Un operación signal elimina un proceso de la lista de procesos esperando, y despierta ese proceso. Las operaciones del semáforo se definen ahora como sigue:

```
wait(S):    S.valor := S.valor – 1;
            if S.valor < 0 then
                        begin
                                    agregar este proceso a S.L;
                                    block;
                        end;
```

```
sgnal(S):   S.valor := S.valor + 1;
            if S.valor <= 0 then
                        begin
                                    remover un proceso P de S.L;
                                    wakeup(P);
                        end;
```

La operación de block suspende el proceso que la invoca. La operación wakeup(P) continúa la operación de un proceso P bloqueado. Estas dos operaciones son proveídas por el sistema operativo en forma de llamadas al sistema.
Note que este tipo de semáforos puede tener valores negativos. En caso de que el valor del semáforo sea negativo, ésta magnitud es el número de procesos que están esperando en el semáforo.
La lista de los procesos que están esperando puede ser fácilmente implementada como un campo de enlace en el PCB de cada proceso. Cada semáforo contiene un valor entero y un puntero a una lista de PCBs. Una forma de agregar y eliminar procesos de la lista, el cual asegura la espera limitada, podría ser una lista de tipo FIFO, donde el semáforo contiene los punteros de la cola y la cabeza. Sin embargo, la lista puede usar cualquier estrategia.
El aspecto critico de los semáforos es que ellos son ejecutados atómicamente. Se debe garantizar que dos procesos no pueden ejecutar las operaciones wait y signal del mismo semáforo al mismo tiempo. Esta situación es un problema de sección critica, y se puede resolver de dos formas.
En un entorno uniprocesador, es decir, donde solo existe una CPU, lo que se hace es prohibir las interrupciones durante el momento que se están ejecutando las operaciones signal y wait. Este esquema trabaja en un entorno de un solo procesador ya que una vez que las interrupciones son prohibidas, las instrucciones de los diferentes procesos no pueden ser entremezcladas. Solo el proceso que esta corriendo actualmente puede ejecutar, hasta que las interrupciones sean habilitadas nuevamente y el scheduler pueda recuperar el control.
En un entorno multiprocesador, la prohibiciones de las interrupciones no funcionan. Las instrucciones de diferentes procesos (corriendo en diferentes procesadores) pueden ser entremezcladas en alguna forma arbitraria. En caso de que el hardware no provea ninguna instrucción especial, podemos emplear alguna de las soluciones de la sección critica que se presentaron anteriormente, donde en este caso la sección critica corresponde de los procedimientos wait y signal.
Se debe tener en cuenta que no se elimina completamente la espera ocupada con éstas últimas definiciones de wait y signal. Más bien, hemos eliminado la espera ocupada de la entrada a las secciones criticas de los programas de aplicación. Además, hemos limitado la espera ocupada a solo las secciones criticas de las

operaciones wait y signal, y éstas secciones son cortas (con un código apropiado no llevaría más de 10 instrucciones). Así, la sección critica casi nunca es ocupada, y la espera ocupada ocurre raramente, y en caso de que ocurra, el tiempo en que ocurre es muy corto. Una situación diferente existe con programas de aplicaciones en los cuales las secciones criticas pueden ser muy largas (minutos u horas) y casi siempre son ocupadas. En este caso, la espera ocupada es extremadamente ineficiente.

Deadlocks e inanición: La implementación de un semáforo con una cola de espera puede resultar en la situación donde dos o más procesos están esperando indefinidamente por un evento que puede ser causado solo por uno de los procesos que esta esperando. El evento en cuestión es la ejecución de una operación signal. Cuando se llega a tal estado, se dice que éstos procesos están en *deadlock*.
Para ilustrar esto, consideremos un sistema consistente de dos procesos, P_0 y P_1, cada uno accediendo a dos semáforos S y Q seteados a 1. Supongamos que P_0 ejecuta wait(S), y luego P_1 ejecuta wait(Q). Cuando P_0 ejecute wait(Q), éste debe esperar hasta que P_1 ejecute signal(Q). Similarmente, cuando P_1 ejecute wait(S), éste debe esperar hasta que P_0 ejecute signal(S). Ya que las operaciones de signal no pueden ser ejecutadas, P_0 y P_1 están en deadlock.

P_0	P_1

wait(S);
wait(Q);
.
.
.
signal(S);
signal(Q);
	wait(Q);
	wait(S);
	.
	.
	.
	signal(Q);
	signal(S);

Se dice que un conjunto de procesos esta en deadlock cuando cada proceso en el conjunto esta esperando por un evento que puede ser causado solo por otro proceso en el conjunto.
Otro problema relacionado con deadlock es el de inanición, el cual es la situación donde los procesos esperan indefinidamente en el semáforo. Puede ocurrir si agregamos y eliminamos procesos de la lista asociados con un semáforo en orden LIFO.

Semáforos binarios: Los semáforos que se construyeron anteriormente se denominan semáforos contadores, ya que el valor del entero puede tomar un valor en un dominio irrestricto. Un semáforo binario es un semáforo donde los valores que puede tomar el entero del semáforo es 0 o 1. A continuación veremos que los semáforos contadores se pueden implementar con semáforos binarios.
Sea S un semáforo contador. Para implementarlo en términos de semáforos binarios se necesitan las siguientes estructuras de datos:

```
var     S1: binary-semaphore;
        S2: binary-semaphore;
        C: integer;
```

Inicialmente, S1 = 1, S2 = 0, y el valor del entero C es seteado al valor inicial del semáforo contador S.
Las operaciones wait y signal del semáforo contador S pueden ser implementadas como sigue:

Wait

```
wait(S1);
C := C – 1;
if C < 0 then
        begin
                signal(S1);
                wait(S2);
        end;
signal(S1);
```

Signal

```
wait(S1);
C := C + 1;
if C <= 0 then  signal(S2);
        else  signal(S1);
```

Problemas clásicos de sincronización

A continuación se verán una serie de problemas de sincronización los cuales son resueltos con semáforos.

Problema del buffer limitado: Consideremos una pileta de n buffers, cada uno capaz de almacenar solo un ítem. El semáforo mutex, el cual provee exclusión mutua al acceso de la pileta de buffer, es inicializado en 1. Los semáforos empty y full cuentan la cantidad de buffers que están vacíos y llenos. El semáforo empty es inicializado en n, mientras que full es inicializado en 0.

El código del proceso productor se ve en la figura 6.12, y el del consumidor en la 6.13. Se podría interpretar como el productor produciendo buffers llenos para el consumidor, o el consumidor produciendo buffers vacíos para el productor.

```
repeat
        ...
        producir un ítem en nextp
        ...
        wait(empty);
        wait(mutex);
        ...
        agregar nextp al buffer
        ...
        signal(mutex);
        signal(full);
until false;
```

Figura 6.12 Estructura del proceso productor.

```
repeat
        wait(full);
        wait(mutex);
        ...
        eliminar un item del buffer a nextc;
        ...
        signal(mutex);
        signal(empty);
        ...
        consumir el item en nextc
        ...
until false;
```

Figura 6.13 Estructura del proceso consumidor.

Problema de los escritores y lectores: Los objetos de dato (tal como un archivo) están para ser compartidos a través de varios procesos concurrentemente. Algunos de estos procesos puede solo querer leer el contenido del objeto compartido, mientras que otros pueden querer cambiarlo (es decir, leerlo y escribirlo). Llamaremos a los procesos que solo quieren leer como lectores, y el resto serán los escritores. Obviamente, si dos lectores acceden al objeto compartido simultáneamente, no surgirá ningún efecto adverso. Sin embargo, si un escritor y algún otro proceso (ya sea escritor o lector) acceden al objeto compartido simultáneamente, se puede producir un caos.
Para asegurarnos que no surgirán estos problemas se requiere que los escritores tengan acceso exclusivo al objeto compartido. La primer idea es que los lectores no tengan que esperar, es decir que si hay un lector esperando es porque un escritor obtuvo permiso de entrar antes de que llegara el lector. Ante esto, la prioridad la tienen los lectores. Una segunda idea es que los escritores no pueden quedarse esperando. Así, si hay un lector en el objeto y existe un escritor esperando, los demás lectores que lleguen deberán esperar a que el escritor realice su trabajo. En esta segunda opción la prioridad es de los escritores.
Note que según lo anterior éste problema puede terminar en inanición. En la primer solución los escritores pueden tener inanición, mientras que en la segunda los lectores. Ante esto se han propuesto otras soluciones. Veremos una solución para el primer caso.
En la solución del primer problema, los lectores comparten la siguientes estructuras de datos:

```
var     mutex, wrt: semaphore;
        readcount: integer;
```

Los semáforos mutex y wrt están inicializados en 1, y readcount es inicializado a 0. El semáforo wrt es común tanto a los procesos lectores como a los escritores. El semáforo mutex es utilizado para asegurar la exclusión mutua cuando la variable readcount esta siendo modificada. La variable readcount mantiene la cantidad de procesos que están actualmente leyendo el objeto. El semáforo wrt funciona como un semáforo de exclusión mutua para los escritores. Este también es utilizado por el último o primer lector que salga o entre en la sección critica. No es usado por los lectores que entren o salgan de la sección critica mientras otros lectores están en sus secciones criticas. El código de los escritores se ve en la figura 6.14, mientras que el de los lectores está en la figura 6.15. Note que si un escritor está en la sección critica y n lectores están esperando, entonces un lector se pone a hacer cola en wrt, y los n-1 lectores se ponen a hacer cola en mutex. Observe también que, cuando un escritor ejecuta signal(wrt), puede continuar la ejecución de los lectores esperando o de solo un escritor. Esta selección es realizada por el scheduler.

```
wait(wrt);
        ...
        se realiza la escritura
        ...
signal(wrt);
```

Figura 6.14 Estructura del proceso escritor.

```
wait(mutex);
        readcount := readcount + 1;
        if readcount = 1 then wait(wrt);
signal(mutex);
        ...
        se realiza la lectura
        ...
wait(mutex);
        readcount := readcount - 1;
        if readcount = 0 then signal(wrt);
signal(mutex);
```

Figura 6.15 Estructura del proceso lector.

Problema de los filósofos cenando: Consideremos 5 filósofos que solo piensan y comen. Los filósofos comparten una mesa donde cada uno tiene una silla. Cuando un filosofo piensa, no interactúa con sus colegas. De vez en cuando un filosofo puede tener hambre por lo que intenta tomar los palos que están a sus costados. Un filosofo puede tomar un palo a la vez. Obviamente, un filosofo no puede tomar un palo que esta en la mano de su vecino. Cuando un filosofo tiene hambre y pudo obtener ambos palillos, puede comenzar a comer sin dejar ningún palillo. Cuando finalizo de comer, pone ambos palillos donde estaban y comienza nuevamente a pensar.
Una solución simple sería representar cada palillo por medio de un semáforo. Un filosofo trata de agarrar un palillo ejecutando la operación wait sobre dicho semáforo, y libera el palillo ejecutando la operación signal sobre el semáforo apropiado. Así, la estructura compartida seria:

```
var palillo: array[0..4] of semaphore;
```

donde todos los elementos del arreglo están inicializados en 1. La estructura del filosofo *i* se muestra en la figura 6.17.

```
repeat
        wait(palillo[i]);
        wait(palillo[i+1mod5]);
        ...
        come
        ...
        signal(palillo[i]);
        signal(palillo[i+1mod5]);
        ...
        piensa
        ...
until false;
```

Figura 6.17 Estructura del filosofo i.

Aunque esta solución garantiza que dos vecinos no estarán comiendo simultáneamente, debe ser rechazado ya que tiene la posibilidad de producir deadlock. Supongamos que a los 5 filósofos al mismo tiempo les agarra hambre y cada uno toma su palillo de la izquierda. Todos los elementos del arreglo palillo son ahora 0. Cuando cada filosofo trate de agarrar su palillo de la derecha provocara que quede esperando por siempre.
Algunas soluciones para que no se produzca tal problema son:

- Permitir que a lo sumo haya 4 filósofos sentados simultáneamente en la mesa.
- Permitir que un filosofo agarre sus palillos solo si ambos palillos (el de su izquierda y derecha) están disponibles (note que ante esto, debe tomarlos en su sección critica).
- Usar una solución asimétrica; esto es, un filosofo impar toma primero su palillo izquierdo y luego el derecho, mientras que un filosofo par tome primero su palillo derecho y luego el izquierdo.

Finalmente, cualquier solución debe garantizar que ningún filosofo entre en inanición. Una solución libre de deadlock no necesariamente elimina la posibilidad de inanición.

Regiones criticas

Aunque los semáforos proveen un conveniente y efectivo mecanismo para la sincronización de procesos, su uso incorrecto puede llevar en errores de tiempo los cuales son difíciles de detectar, ya que éstos errores ocurren cuando alguna secuencia de ejecución en particular toma lugar, y éstas secuencias no siempre ocurren.
Para ilustrar un ejemplo de estos errores, veamos una solución al problema de la sección critica llevando a cabo una solución con semáforos. Todos los procesos comparten una variable semáforo mutex, el cual esta inicializada en 1. Cada proceso debe ejecutar wait(mutex) antes de entrar a la sección critica, y signal(mutex) al salir. Si la secuencia no es observada, dos procesos pueden estar en sus secciones criticas simultáneamente. Veamos las diferentes dificultades que pueden resultar. Además, note que éstas dificultades surgirán con un único proceso que no funcione bien. Estas situaciones pueden ocurrir por simples errores de programación o por un mal desempeño del programador.

- Supongamos que un proceso intercambia el orden en el cual las operaciones wait y signal son ejecutadas sobre el semáforo mutex, resultando:

```
signal(mutex);
...
sección critica
...
wait(mutex);
```

  En esta situación, varios procesos pueden estar ejecutando en sus secciones criticas simultáneamente, violando el requerimiento de exclusión mutua.

- Supongamos que un proceso reemplaza signal(mutex) por \wait(mutex). Esto es, ejecuta:

```
wait(mutex);
...
sección critica
...
wait(mutex);
```

  En este caso puede ocurrir un deadlock.

- Supongamos que un proceso omite signal(mutex), o wait(mutex), o ambos. En este caso, o la exclusión mutua es violada, o puede ocurrir un deadlock.

Para tratar con dichos errores, se verán un número de constructores de alto nivel. En este capítulo se vera uno que es fundamental con la sincronización, el constructor *critical region*. Más adelante se vera otro constructor llamado *monitor*. En ambos constructores se supone que un proceso se compone de un dato local y de un programa secuencial que puede operar sobre el dato. Al dato local solo se puede acceder por medio del programa secuencial, es decir, no se puede acceder a un dato local de un proceso por medio de otro proceso. Sin embargo, los procesos pueden compartir datos globales.
Este constructor requiere que una variable v de tipo T, el cual está para ser compartida por varios procesos, es declarada como:

```
var v: shared T;
```

La variable v puede ser accedida solo dentro de la sentencia *region* de la siguiente forma:

region v **when** B **do** S;

Éste constructor significa que, mientras la sentencia S esta siendo ejecutada, ningún otro proceso puede acceder a la variable v. La expresión B es una expresión booleana que gobierna el acceso a la sección critica. Cuando un proceso trata de entrar a la region de la sección critica, se evalúa la expresión B. En caso de que sea verdadera se ejecuta la sentencia S. Si es falsa, el proceso abandona la exclusión mutua y es demorado hasta que B sea verdadera y ningún otro proceso esta en la region asociada con v. Así, si las dos sentencias:

region v **when** true **do** S_1;
region v **when** true **do** S_2;

están ejecutando concurrentemente en procesos secuenciales distintos, y el resultado será equivalente a ejecutar secuencialmente "S_1 seguido de S_2" o "S_2 seguido de S_1".
El constructor critical-region puede ser efectivamente utilizado para resolver ciertos problemas generales de sincronización. Para ver esto, veamos el código del problema del buffer limitado. El espacio del buffer y sus punteros son encapsulados en:

```
var buffer: shared record
        pool: array[0..n-1] of item;
        count, in, out: integer;
            end;
```

El proceso productor inserta un nuevo ítem en nextp ejecutando el código que se ve abajo, mientras que el consumidor remueve un ítem del buffer y lo pone en nextc ejecutando también su código correspondiente.

```
region buffer when count < n do
        begin
                pool[in] := nextp;
                in := in + 1 mod n;
                count := count + 1;
        end;
```

Código del productor

```
region buffer when count > 0 do
        begin
                nextc := pool[out];
                out := out + 1 mod n;
                count := count - 1;
        end;
```

Código del consumidor

Monitores

Otro tipo de constructor de sincronización de alto nivel es el *monitor*. Su representación consiste de la declaración de las variables, y luego el cuerpo de los procedimientos o funciones que operan sobre éstos tipos. Su sintaxis se ve en el cuadro.

```
type monitor-name = monitor
        declaracion de variables

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

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

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

        begin
        codigo de inicializacion
        end.
```

Un procedimiento definido en un monitor puede acceder solo a las variables declaradas localmente en el monitor y los parámetros formales. Similarmente, las variables locales de un monitor pueden ser accedidas sólo por los procedimientos locales.
El constructor monitor asegura que sólo un proceso a la vez puede estar activo en el monitor. Consecuentemente, los programadores no necesitan codificar explícitamente ésta restricción de sincronización. Sin embargo, el constructor monitor, como se definió, no asegura algunos esquemas de sincronización. Para esto, se necesita definir mecanismos adicionales de sincronización. Estos mecanismos son proveídos por el constructor *condition*. Un programador que necesita escribir su propio esquema de sincronización puede definir una o más variables del tipo condition:

var x, y: condition;

Las únicas operaciones que pueden ser invocadas sobre variables del tipo condition son wait y signal. La operación *x.wait;* significa que el proceso invocando esta operación es suspendido hasta que otro proceso invoque *x.signal;*.
La operación x.signal continúa solo un proceso suspendido. Si no existen procesos suspendidos, la operación signal no tiene efecto, es decir, es como si fuera que nunca fue ejecutada (Figura 6.20). Note el contraste de esta operación con la operación asociada signal para semáforos, la cual siempre afecta el estado del semáforo.

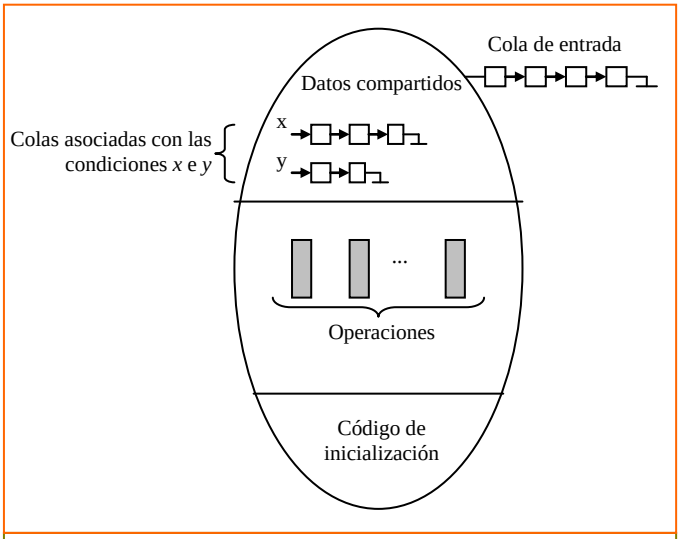


Figura 6.20 Monitor con variables de condición.

Ahora supongamos que, cuando la operación x.signal es invocada por un proceso P, existe un proceso Q suspendido asociado con la condición x. Claramente, si se le permite al proceso suspendido Q continuar su ejecución, entonces el proceso que invoco la operación signal, es decir, P, debe esperar. De otra manera, tanto P y Q estarán ejecutando simultáneamente en el monitor. Note, sin embargo, que ambos procesos pueden continuar conceptualmente su ejecución. Existen dos posibilidades:

1. P espera hasta que Q deje el monitor, o espera por otra condición.
2. Q espera hasta que P deje el monitor, o espera por otra condición.

Ante el hecho de que P ya esta ejecutando en el monitor, pareciera que la segunda opción es más razonable. El lenguaje Pascal lo que hace es que cuando P ejecuta la operación signal, éste debe inmediatamente dejar el monitor y es Q quien es continuado.
Veamos una solución libre de deadlock para el problema de los filósofos hambrientos. Recordemos que un filosofo solo puede tomar los palillos en caso de que ambos estén disponibles en la mesa. Para codificar la solución, distingamos cuales son los posibles estados en las que puede caer un filosofo. Para este propósito, introduzcamos la siguiente estructura de datos:

```
var estate: array[0..4] of (thinking, hungry, eating);
```

Así, el filósofo i puede setear state[i] = eating solo si sus dos vecinos no están comiendo (state[i+4 **mod** 5] != eating **and** state[i+1 **mod** 5] != eating).
También necesitamos declarar:

```
var self: array[0..4] of condition;
```

donde el filosofo i puede esperar cuando tenga hambre, pero no pudo obtener sus dos palillos.
Comencemos con la descripción de la solución. La distribución de los palillos esta controlada por el monitor *dp*, el cual es una instancia del tipo monitor *dining-philosophers*, cuya definición se ve en la figura 6.21.

Cada filosofo antes de comenzar a comer, debe invocar a la operación *pickup*. Esta puede resultar en la suspensión del proceso filosofo. Luego de que se complete con éxito la operación, el filosofo puede comer. Luego de esto, el filosofo invoca la operación *putdown* y comienza a pensar. Así, el filosofo i debe invocar la operación pickup y putdown en la siguiente secuencia:

```
dp.pickup(i);
...
come
...
dp.putdown(i);
```

Con ésta solución de seguro dos filósofos no estarán comiendo simultáneamente ni se producirá deadlock. Notemos, sin embargo, que es posible que un filosofo entre en inanición.

```
type dining-philosophers = monitor
        var state: array[0..4] of (thinking, hungry, eating);
        var self: array[0..4] of condition;

        procedure entry pickup(i: 0..4);
                begin
                        state[i] = hungry;
                        test(i);
                        if (state[i] != eating) then self[i].wait;
                end;

        procedure entry putdown(i: 0..4);
                begin
                        state[i] = thinkimg;
                        test(i + 4 mod 5);
                        test(i + 1 mod 5);
                end;

        procedure test(k: 0..4);
                begin
                        if state[k + 4 mod 5] != eating and state[k] = hungry and state[k + 1 mod 5] != eating then
                                begin
                                        state[k] := eating;
                                        self[k].signal;
                                end;
                end;

        begin
                for i:= 0 to 4 do state[i] := thinking;
        end.
```

Figura 6.21 Solución con un monitor al problema de los filósofos hambrientos.

Consideremos ahora una solución con el mecanismo de monitor utilizando semáforos. Para cada monitor se tiene un semáforo mutex (inicializado en 1). Un proceso debe ejecutar wait(mutex) antes de entrar al monitor, y debe ejecutar signal(mutex) luego de dejar el monitor.
Ya que un proceso que ejecuta la operación signal debe esperar hasta que el proceso continuado salga o espere, un semáforo adicional *next* es introducido, inicializado en 0, sobre el cual los procesos que ejecutan la operación signal pueden suspenderse a ellos mismos. Una variable entera next-count cuenta el número de procesos suspendidos sobre next. Así, cada procedimiento externo F será reemplazado por:

```
wait(mutex);
...
cuerpo de F
...
if next-count > 0 then signal(next);
                  else signal(mutex);
```

La exclusión mutua esta asegurada por la utilización del monitor.
A continuación se describe como son implementadas las variables de condición. Para cada condición x introducimos un semáforo x-sem y una variable entera x-count, ambas inicializadas en 0. La operación x.wait y x.signal son implementadas como:

```
x-count := x-count + 1;
if next-count > 0 then signal(next);
                  else signal(mutex);
wait(x-sem);
x-count := x-count – 1;
```

x.wait

```
if x-count > 0 then
        begin
                next-count := next-count + 1;
                signal(x-sem);
                wait(next);
                next-count := next-count – 1;
        end;
```

x.signal

Retornemos al asunto de monitores. Si varios procesos están suspendidos sobre la condición x, y se ejecuta una operación x.signal por algún proceso, para saber cual de todos los procesos suspendidos debe ser continuado se podría utilizar un orden FCFS, por lo que el proceso que más tiempo hace que esta suspendido es el que será continuado. Sin embargo, existen circunstancias en las cuales se necesita un esquema de scheduling más adecuado. Para éste propósito se utiliza el constructor *conditional-wait* que tiene la forma:

x.wait(c);

donde c es una expresión entera que es evaluada en el momento que se ejecuta la operación wait. El valor de c, el cual es llamado número de prioridad, es luego almacenado con el nombre del proceso que esta suspendido. Cuando x.signal es ejecutada, el proceso con el número de prioridad más pequeño asociado con el número de prioridad c, es el que se continuara.

Para ver como funciona, consideremos el monitor que se ve en la figura 6.22, el cual controla la asignación de un único recurso entre varios procesos. Cada proceso, cuando pide por la asignación de su recurso, especifica el máximo tiempo que planea usar el recurso. El monitor asigna el recurso al proceso que tiene el tiempo de uso más corto.

```
type resource-allocation = monitor
        var busy: boolean;
        var x: condition;

        procedure entry acquire(time: integer);
                begin
                        if busy then x.wait(time);
                        busy := true;
                end;

        procedure entry release;
                begin
                        busy := false;
                        x.signal;
                end;

        begin
                busy := false;
        end.
```

Figura 6.22 Un monitor que asigna un único recurso.

Un proceso que necesita acceder al recurso en cuestión debe realizar la siguiente secuencia:

```
R.acquire(t);
...
acceso al recurso;
...
R.release;
```

donde R es una instancia de resource-allocation.

Desafortunadamente, la estructura monitor no puede garantizar que:

- Un proceso puede acceder al recurso sin primero ganar el permiso al recurso.
- Un proceso puede que nunca libere el recurso una vez que se le ha sido concedido el acceso.
- Un proceso puede intentar liberar un recurso que nunca ha pedido.

- Un proceso puede pedir el mismo recurso dos veces (sin liberar el recurso del primer pedido).

Las mismas dificultades se encuentran en el constructor de la sección critica.

Sincronización en Solaris 2

Antes de que saliera Solaris 2, SunOS usaba secciones criticas para proteger estructuras de datos importantes. Solaris 2 fue diseñado para soportar computación en tiempo real, por medio de múltiple threads, y soportar multiprocesos. Continuar con el uso de secciones criticas podría ocurrir un alta decaimiento del rendimiento en éstos sistemas. Además, las secciones criticas no se podrían implementar vía interrupciones, ya que las interrupciones podrían ocurrir en otros procesadores en un sistema multiprocesador. Para evitar estos problemas, Solaris 2 usa *mutex adaptative* para proteger el acceso a cada ítem de dato critico.

En un sistema multiprocesador, un mutex adaptative comienza como un semáforo estándar implementado como un spinlock. Si el dato esta bloqueado, y por lo tanto en uso, el mutex adaptative realiza una de dos opciones. Si el bloqueo esta mantenido por un thread que esta actualmente corriendo, el thread que pidió el dato espera para que el bloqueo se desbloquee ya que se espera que el thread que esta manteniendo el bloqueo lo libere pronto. Si el thread manteniendo el bloqueo no esta actualmente en estado de corriendo, el thread que pidió el dato se bloquea, haciendo que se duerma hasta que es despertado por la liberación del bloqueo. Este es puesto a dormir para evitar que de vueltas cuando el bloqueo no se libere en un tiempo razonablemente rápido (ya que el thread que tiene el bloque no esta corriendo). En un sistema uniprocesador, el thread manteniendo el bloqueo nunca esta corriendo si la traba esta siendo testeada por otro thread, ya que puede correr solo un thread a la vez. Por lo tanto, en un sistema uniprocesador, los thread siempre duermen en lugar de dar vueltas si ellos encuentran una traba.

En situaciones de sincronización más complejas, Solaris 2 utiliza variables de condición y bloqueos del tipo lectores-escritores. El método del mutex adaptativo descripto anteriormente es usado para proteger aquellos datos que son accedidos por segmentos de código cortos. Es decir, un mutex es usado si la traba será mantenida por menos de cientos de instrucciones. En caso de que el segmento de código sea mayor, la espera de dar vueltas seria muy ineficiente. Ante esto, se utilizan las variables de condición. Si el bloqueo que se desea hacer ya lo ha realizado otro, entonces el thread que pidió hacer el bloqueo emite un wait y se duerme. Cuando el thread que tenia el bloqueo lo libera, éste emite un signal al siguiente thread durmiendo en la cola. Los costos extras de poner un thread a dormir y luego despertarlo, y los asociados al context switch, son menores que el costo de gastar varios cientos de instrucciones en una espera de dar vueltas.

Los bloqueos de escritores-lectores son usados para proteger los datos que son accedidos frecuentemente, pero usualmente de manera de solo lectura. En esta circunstancia, los bloqueos lectores-escritores son más eficientes que los semáforos, ya que múltiple threads pueden estar leyendo un dato concurrentemente, mientras que los semáforos siempre serializarán el acceso a los datos. Los bloqueos lectores-escritores son muy costosos de implementar, por lo que nuevamente son usados solo en grandes secciones de código.

Transacciones atómicas

La exclusión mutua de secciones criticas asegura que las secciones criticas son ejecutadas atómicamente. Esto es, si dos secciones criticas son ejecutadas concurrentemente, el resultado es equivalente a sus ejecuciones en cualquier orden. Aunque ésta propiedad es muy útil en muchas aplicaciones, existen otras donde queremos que la sección critica forme una única unidad lógica de trabajo donde las operaciones son realizadas o todas o no se hace ninguna de ellas. Un ejemplo es en una transferencia, en donde una cantidad es debitada y otra es acreditada. Claramente, es esencial para la consistencia de los datos, o que se haga la acreditación y el débito, o ninguno de los dos.

Esta sección está relacionada con los sistemas de base de datos. Las base de datos están involucradas con el almacenamiento y recuperación de datos, y con la consistencia de los datos.

Modelo del sistema: una colección de instrucciones (operaciones) que realizan una única función lógica es llamada transacción. El mayor problema en el procesamiento de transacciones es preservar la atomicidad a pesar de fallas en la computadora. En esta sección se verán varios métodos que aseguran la atomicidad de la transacción.
Una transacción es una secuencia de operaciones **read** y **write**, terminadas por su operación **commit** o **abort**. Una operación commit significa que la transacción ha terminado su ejecución exitosamente, mientras que una operación abort significa que la transacción a parado su normal ejecución debido a algún error. El efecto de una transacción finalizada exitosamente no puede ser desasida por la cancelación de la misma.
Una transacción puede parar su normal ejecución debido a una falla del sistema. En ambos casos, ya que una transacción abortada puede haber cambiado algunos de los datos antes de que sea parada, el estado de estos datos puede no ser el mismo que hubiera quedado en caso de que la transacción haya terminado exitosamente. Si se asegura la propiedad de atomicidad de la transacción, una transacción abortada no debe tener efecto sobre los datos que ya ha modificado. Así, el estado de los datos accedidos por una transacción abortada debe ser restablecido al estado que estaban los datos antes de que la transacción comenzara su ejecución. Si esto ocurre, se dirá que una transacción ha sido roll back.
Para determinar como se puede asegurar la propiedad de atomicidad, primero se deben identificar las propiedades de los dispositivos usados para almacenar los datos accedidos por las transacciones. Varios tipos de medios de almacenamiento se distinguen por sus velocidades, capacidad, y por su elasticidad en caso de fallos.

- *Almacenamiento volátil:* la información residente en el almacenamiento volátil no sobrevive a caídas del sistema. Ejemplos de tales almacenamientos son la memoria principal y la memoria cache. El acceso a la memoria volátil es muy rápido, ya que es una memoria de alta velocidad y porque es posible acceder a cualquier dato directamente.
- *Almacenamiento no volátil:* la información residente en el almacenamiento no volátil usualmente sobrevive a caídas del sistema. Ejemplos de tales medios de almacenamientos son discos y cintas magnéticas. Actualmente, el almacenamiento no volátil es mucho más lento que el almacenamiento volátil ya que los discos y las cintas son electrónicas y requieren movimientos físicos.
- *Almacenamiento estable:* la información residente en almacenamiento estable nunca se pierde. Para implementar una aproximación a tales almacenamientos, se necesita reproducir la información en varios medios de almacenamientos no volátiles (usualmente discos) con modos de fallas diferentes, y cambiar la información de una manera controlada.

Recuperación basada en registros: una forma de asegurar la atomicidad es registrar, en almacenamiento estable, información describiendo todas las modificaciones realizadas por la transacción. El sistema mantiene, en un almacenamiento estable, una estructura de datos llamada *log*. Cada registro del log describe una única operación de una transacción write, y tiene los siguientes campos:

- *Nombre de la transacción:* el nombre de la transacción que realiza la operación write.
- *Nombre del ítem de dato.*
- *Valor viejo:* el valor del dato antes de la escritura.
- *Valor nuevo:* el valor que el dato tendrá después de la escritura.

Antes de que una transacción T_i comience su ejecución, el registro <T_i starts> es escrito en el log. Durante su ejecución, cualquier operación de escritura realizada por la transacción T_i es precedida por la escritura de un nuevo registro apropiado en el log. Cuando T_i commits, el registro <T_i commits> es escrito en el log.
Ya que la información en el log es usada para reconstruir el estado del dato, no podemos permitir que los actuales cambios realizados en el dato tomen lugar antes de que los correspondientes registros del log estén escrito en el almacenamiento estable. Ante esto, antes de que se ejecute, por ejemplo, la operación write(X), el correspondiente registro log debe ser escrito en el almacenamiento estable.

Note que se requieren dos escrituras físicas para cada pedido de escritura lógica. Además, se necesita más almacenamiento: para los datos mismos y para los registros del log. Usando el log, el sistema puede manejar cualquier fallo. El algoritmo de recuperación usa dos procedimientos:

- Undo(T_i), el cual reestablece el valor de todos los datos cambiados por la transacción T_i al los viejos valores.
- Redo(T_i), el cual setea el valor de todos los datos modificados por la transacción T_i a los nuevos valores.

Si una transacción T_i aborta (es decir, tiene el registro <T_i starts> al inicio, pero no el registro <T_i commits> al final), entonces se puede restaurar el estado de los datos que se han cambiado ejecutando simplemente undo(T_i).

Checkpoints: cuando ocurre una falla de sistema, se debe consultar el log para determinar aquellas transacciones que se necesitan volver a hacer y aquellas que se necesitan des hacer. En principio, se necesita recorrer en el log completo para hacer estas determinaciones. Existen dos inconvenientes para esto:

1. El proceso de búsqueda esta consumiendo tiempo.
2. La mayoría de las transacciones que, acorde a sus algoritmos, necesitan ser re hechas, ya han cambiado los datos que actualmente el log dice que necesita modificar. Aunque re hacer las modificaciones de los datos no causara daño, esta acción tomara tiempo.

Para reducir este tipo de overhead, se introduce el concepto de punto de chequeos. Durante la ejecución, el sistema mantiene el log y la idea de escribir primero el registro en el log en caso de una escritura y luego ejecutar la escritura. Además, el sistema realiza periódicos puntos de chequeo, el cual requiere la siguiente secuencia de acciones:

1. Copiar todos los registros del log actualmente residentes en almacenamiento volátil (usualmente en memoria principal) a almacenamiento estable.
2. Copiar todos los datos modificados residentes en almacenamiento volátil a almacenamiento estable.
3. Copiar un registro en el log <checkpoint> en almacenamiento estable.

La presencia de un registro <checkpoint> en el log permite al sistema llevar pista para usarlo en el procedimiento de recuperación. Consideremos una transacción T_i que commited antes de punto de chequeo. El registro <T_i commits> aparecerá en el log antes del registro <checkpoint>. Cualquier modificación realizada por T_i debe haber sido escrita en el almacenamiento estable o antes del punto de chequeo, o como parte del punto de chequeo mismo (es decir, cuando se produjo el punto de chequeo, por la rutina misma del chequeo se copia todo en almacenamiento estable). Así, en el momento de recuperación, no existe necesidad de realizar la operación redo sobre T_i.
Ante esto, el nuevo algoritmo de recuperación es: luego de que ocurre una falla, la rutina de recuperación examina el log para determinar la transacción T_i más reciente que comenzó su ejecución antes del punto de chequeo más reciente. Este encuentra tal transacción buscando en el log hacia atrás para encontrar el primer registro <checkpoint>, y luego encuentra el subsecuente registro <T_i start> (es decir, la transacción empezó pero no llego a terminar).
Una vez que la transacción T_i a sido identificada, las operaciones redo y undo necesitan ser aplicadas solo a la transacción T_i y toda transacción T_j que comenzó la ejecución después de la transacción T_i. Denotaremos este conjunto de transacciones por medio del conjunto T. El resto del log puede entonces ser ignorado. Las operaciones de recuperación que se requieren son las siguientes:

- Para toda transacción T_k en T tal que el registro <T_k commits> aparece en el log, ejecutar redo(T_k).
- Para toda transacción T_k en T tal que el registro <T_k commits> no aparece en el log, ejecutar undo(T_k).

Transacciones atómicas concurrentes: ya que cada transacción es atómica, la ejecución concurrente de transacciones debe ser equivalente al caso donde estas transacciones se ejecutan serialmente en algún orden arbitrario. Esta propiedad, llamada seriabilidad, se puede mantener simplemente ejecutando cada transacción en una sección critica. Esto es, todas las transacciones comparten un semáforo común mutex, el cual se inicializa en 1. Cuando una transacción comienza la ejecución, su primer acción es ejecutar wait(mutex). Luego de que la transacción hace un commit o un abort, ejecuta signal(mutex).
Aunque este esquema asegura la atomicidad de las transacciones ejecutándose concurrentemente, es demasiado restricto. Como veremos, existen muchos casos donde se permitirá solapar la ejecución de transacciones, mientras se mantenga la seriabilidad. Hay un número diferente de algoritmos de control de concurrencia para asegurar la seriabilidad.

- **Serializabilidad:** Considere un sistema con dos ítems A y B, ambos son leídos y escritos por dos transacciones T_0 y T_1. Supongamos que estas transacciones son ejecutadas atómicamente en el orden T_0 seguida de T_1. Esta secuencia de ejecución, el cual es llamada un schedule, se ve en la figura 6.23.

T_0	T_1
Read(A)	
Write(A)	
Read(B)	
Write(B)	
	Read(A)
	Write(A)
	Read(B)
	Write(B)

Figura 6.23 Schedule 1: un schedule serial.

Un schedule donde cada transacción es ejecutada atómicamente es llamado schedule serial. Cada schedule serial consiste de una secuencia de instrucciones de varias transacciones donde las instrucciones pertenecientes a una transacción aparecen juntas en el scheduler. Así, para un conjunto de n transacciones, existen n! diferentes schedulers seriales validos.
Si permitimos que dos transacciones solapen su ejecucion, entonces el scheduler resultante no es serial. Un scheduler no serial no necesariamnete implica que su resultado sea una ejecución incorrecta. Para ver esto, definamos la noción de operaciones conflictivas. Consideremos un scheduler S en el cual hay dos operaciones consecutivas O_i y O_j de las transacciones T_i y T_j respectivamente. Diremos que O_i y O_j están en conflicto si acceden al mismo dato y, al menos una de estas operaciones es la operación write. Para ilustrar este concepto, consideremos el scheduler no serial que se ve en la figura 6.24. La operación write(A) de T_0 esta en conflicto con la operación read(A) de T_1. Sin embargo, la operación write(A) de T_1 no tiene conflicto con la operación read(B) de T_0, ya que las dos operaciones acceden a ítems diferentes.

T_0	T_1
Read(A)	
Write(A)	
	Read(A)
	Write(A)
Read(B)	
Write(B)	
	Read(B)
	Write(B)

Figura 6.24 Schedule 2: Un schedule serializable concurrente.

Ilustremos la idea de swapping considerando nuevamente el segundo scheduler. Como la operación write(A) de T_1 no tiene conflicto con la operación read(B) de T_0, podemos intercambiar estas operaciones para generar un scheduler equivalente. Sin tener en cuenta el estado inicial del sistema, ambos schedulers producen el mismo estado final. Continuando con este procedimiento de cambio de operaciones sin conflictos:

- Cambiar la operación read(B) de T_0 con la operación read(A) de T_1.
- Cambiar la operación write(B) de T_0 con la operación write(A) de T_1.
- Cambiar la operación write(B) de T_0 con la operación read(A) de T_1.

- **Protocolo de bloqueo:** una forma de asegurar la seriabilidad es la de asociar a cada ítem de dato un bloqueo, y requerir que cada transacción siga un protocolo de bloqueo que gobierna como los bloqueos son adquiridos y liberados. Hay varios modos en el cual el dato puede ser bloqueado. En esta parte, mostraremos dos modos:
  - *Compartido:* si una transacción T_i ha obtenido un bloqueo de modo compartido (denotado por S) sobre el dato Q, entonces T_i puede leer este ítem, pero no puede escribirlo.
  - *Exclusivo:* si una transacción T_i ha obtenido un bloqueo de modo exclusivo (denotado por X) sobre el dato Q, entonces T_i puede tanto leer como escribir el ítem.

  Se requiere que cada transacción pida un bloqueo de un modo apropiado sobre el dato Q, dependiendo del tipo de operación que se realizara sobre Q.
  Para acceder al dato Q, la transacción T_i debe primero bloquear Q en el modo apropiado. Si Q no esta actualmente bloqueado, entonces se concede el bloqueo, y T_i puede acceder al dato. Sin embargo, si el dato Q esta actualmente bloqueado, por alguna otra transacción, entonces T_i debe esperar. Más precisamente, supongamos que T_i pide un bloqueo exclusivo sobre Q. En este caso, T_i debe esperar hasta que el bloqueo sobre Q se libera. Si T_i pide un bloqueo compartido sobre Q, entonces T_i debe esperar si Q esta bloqueado en modo exclusivo. De otra manera, este obtendrá el bloqueo y acceso a Q.
  Un protocolo que asegura seriabilidad es el protocolo de bloqueo de dos fases. Este protocolo requiere que cada transacción emita pedidos de bloqueo y desbloqueo en dos fases:
  - *Fase de aumento:* una transacción puede obtener bloqueos, pero no puede liberar ningún bloqueo.
  - *Fase de disminución:* una transacción puede liberar bloqueos, pero no puede obtener nuevos bloqueos.

  Inicialmente, una transacción esta en la fase de crecimiento. La transacción adquiere bloqueos cuando los necesita. Una vez que la transacción libera un bloqueo, entra en la etapa de disminución, y no se pueden emitir más pedidos de bloqueos.
  El protocolo de bloqueos de dos fases asegura la seriabilidad de conflictos pero no asegura libertad de deadlock.

7. Deadlocks

En un entorno multiprogramado, varios procesos pueden competir por recursos. Un proceso pide recursos; en caso de que todos los recursos que pida no están todos disponibles a la vez, entonces entra en estado de espera. Pero esto puede provocar que el proceso nunca salga de su estado de espera, ya que los recursos que pidió están siendo utilizados por otros procesos que también están en estado de espera. A ésta situación se la denomina deadlock.

En este capítulo se verán métodos que tratan los problemas de deadlock. Note, sin embargo, que los sistemas operativos más actuales no proveen la prevención de deadlock. Tales métodos se están agregando en los sistemas ya que los problemas de deadlock se están haciendo cada vez más comunes. Esto es causado porque día a día esta aumentando el número de procesos, el aumento de recursos en un sistema (incluyendo CPUs), y la tendencia hacia los sistemas de base de datos en lugar de los sistemas batch.

Modelo del sistema

Un sistema consiste de un número de recursos que son distribuidos a través de varios procesos que están compitiendo por los mismos. Los recursos son particionados en varios tipos, tales como espacio de memoria, ciclos de CPU, archivos, dispositivos de I/O (drives de impresoras o de cinta). Si un sistema tiene dos CPUs, entonces el tipo de recurso CPU tiene dos instancias.

Si un proceso pide una instancia de un tipo de recurso, la asignación de cualquier instancia del tipo satisfará el pedido. En caso contrario, las instancias de un mismo tipo no son idénticas, y las clases de tipos de recursos no han sido bien definidas. Por ejemplo, supongamos que se tienen dos impresoras, una blanco y negro y la otra color. Se podría pensar que ambas pertenecen a un mismo tipo de recurso que es impresora. Sin embargo, si un usuario desea imprimir en color y se le otorga la de blanco y negro, surgirá un problema. Ante esto, en realidad ambas impresoras no pertenecen al mismo tipo y deben ser separadas en dos tipos de recursos.

Un proceso debe pedir un recurso antes de usarlo. Un proceso puede pedir la cantidad de recursos que necesite para llevar a cabo su tarea. Obviamente, el número de recursos pedidos no puede ser mayor a la cantidad de recursos que hay disponibles.

Bajo una normal operación, un proceso puede utilizar un recurso solo en la siguiente secuencia:

1. **Pedido:** en caso de que el pedido no pueda ser otorgado inmediatamente (por ejemplo, porque el recurso esta siendo utilizado por otro proceso), entonces el proceso que pide debe esperar hasta que pueda adquirir el recurso.
2. **Uso:** el proceso puede operar sobre el recurso (por ejemplo, si el recurso es una impresora, el proceso puede imprimir sobre la impresora).
3. **Liberación:** el proceso libera el recurso.

El pedido y la liberación del recurso son llamadas al sistema. Algunos ejemplos de llamadas son **request** y **release device**, **open** y **close file**, y **allocate** y **free memory**. El pedido y liberación de otros recursos se puede llevar a cabo a través de operaciones signal y wait sobre semáforos. Por lo tanto, para cada uso, el sistema operativo chequea para estar seguro que el proceso que esta usando el recurso lo ha pedido y se le ha sido asignado. Una tabla de sistema registra si un recurso esta libre o asignado, y, en caso de estar asignado, a que proceso. En caso de que un proceso pida un recurso que esa asignado a otro proceso, éste puede ser agregado a la cola de procesos esperando por ese recurso.

Un conjunto de procesos están en estado de deadlock cuando cada proceso en el conjunto esta esperando por un evento que puede ser causado sólo por un proceso del conjunto. Los recursos pueden ser físicos (por ejemplo, impresoras, drives de cinta, espacio de memoria, y ciclos de CPU), o recursos lógicos (por ejemplo, archivos, semáforos y monitores). Sin embargo, otros tipos de eventos pueden resultar en deadlocks (por ejemplo, la facilidad IPC vista en el capítulo 4).

Caracterización del deadlock

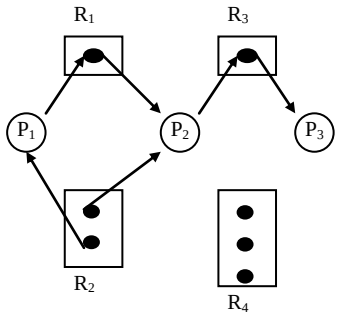
En un deadlock, los procesos nunca finalizan la ejecución, y los recursos del sistema nunca se liberaran, provocando que otros trabajos nunca comiencen. A continuación se verán las características de un deadlock y los métodos para tratarlo.

Condiciones necesarias: un deadlock surge en caso de que las 4 condiciones siguientes se cumplan todas a la vez.

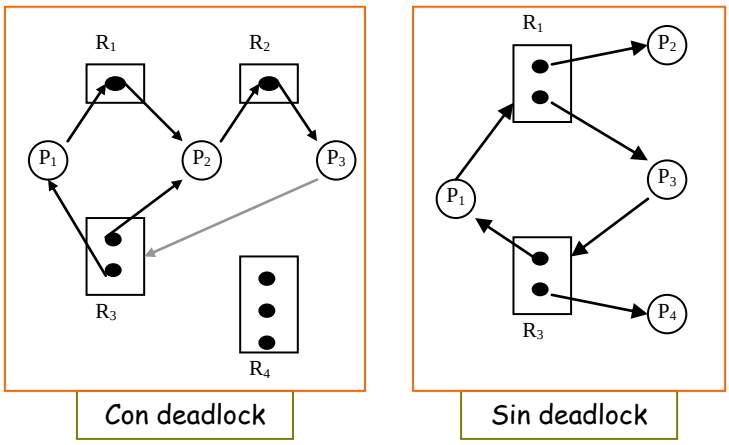
1. *Exclusión mutua:* por lo menos un recurso se debe mantener en modo no compartido, es decir, solo un proceso a la vez puede estar usando el recurso. En caso de que otro proceso pida el recurso, el proceso que pide debe ser demorado hasta que se libere el recurso.
2. *Agarrar y esperar:* debe existir un proceso que tiene en su poder al menos un recurso y esta esperando para adquirir un recurso adicional el cual esta siendo usado por otro proceso.
3. *Sin desalojo:* los recursos no pueden ser desalojados, es decir, un recurso puede ser liberado solo voluntariamente por el proceso que lo mantiene, luego de que el proceso haya completado su tarea.
4. *Espera circular:* debe existir un conjunto $\{P_0, P_1, P_2, ..., P_n\}$ de procesos esperando tal que el proceso P_0 esta esperando por un recurso que esta usando P_1, P_1 esta esperando por un recurso que tiene P_2, ..., P_{n-1} esta esperando por un recurso que tiene P_n y P_n esta esperando por un recurso que tiene P_0.

Grafo de asignación de recursos: los deadlocks se pueden describir por lo que se llama grafo de asignación de recursos. Este grafo consiste de un conjunto de vértices y de un conjunto de arcos. Los vértices se dividen en dos grupos diferentes de nodos $P=\{P_1, P_2, ..., P_n\}$, el cual es el conjunto de todos los procesos activos del sistema, y $R=\{ R_1, R_2, ..., R_m\}$, el conjunto de todos los tipos de recursos del sistema.
Un arco dirigido desde el proceso P_i al recurso de tipo R_j significa que el proceso P_i pidió una instancia de un recurso del tipo R_j y esta esperando actualmente por dicho recurso (arco de pedido). Un arco dirigido desde un tipo de recurso R_j al proceso P_i significa que la instancia del recurso del tipo R_j ha sido asignada al proceso P_i (arco de asignación). Se representara cada proceso como un circulo y cada tipo de recurso como un cuadrado. Como de un tipo de recurso puede haber más de una instancia, se representa dentro del cuadrado con círculos negros las instancias de dicho tipo de recurso.
Cuando un proceso pide una instancia de algún tipo de recurso, se inserta en el grafo un arco de pedido. Cuando dicho pedido puede ser cumplido, se transforma dicho arco en una arco de asignación. Cuando el proceso ya no necesita el recurso, entonces el arco de asignación es eliminado.
Ejemplo:

- Conjuntos P, R y E:
  - $P = \{P_1, P_2, P_3\}$
  - $R = \{R_1, R_2, R_3, R_4\}$
  - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- Instancias del recurso:
  - Una instancia del recurso del tipo R_1.
  - Dos instancias del recurso del tipo R_2.
  - Una instancia del recurso del tipo R_3.
  - Tres instancias del recurso del tipo R_4.
- Estados de los procesos:
  - El proceso P_1 esta manteniendo una instancia del recurso de tipo R_2, y esta esperando por una instancia del recurso de tipo R_1.
  - El proceso P_2 esta manteniendo una instancia del tipo R_1 y R_2, y esta esperando por una instancia del recurso de tipo R_3.
  - El proceso P_3 esta manteniendo una instancia de R_3.



Ante este dibujo, en caso de que no exista ciclo en el grafo entonces de seguro no existe deadlock. En caso de que haya un ciclo puede haber entonces deadlock. En caso de que cada tipo de recurso tenga una instancia entonces la presencia de un ciclo implica directamente la presencia de deadlock. En caso de que un tipo de recurso implicado en el ciclo tiene varias instancias entonces la presencia de un ciclo no implica deadlock. En este caso, la presencia del ciclo en el grafo es una condición para que exista deadlock pero no es la única condición.
A continuación se presentan dos ejemplos, uno con la existencia de deadlock y el otro no.
En el primer ejemplo el sistema esta en deadlock ya que P3 nuca podrá obtener la instancia del recurso del tipo R2. En el segundo ejemplo el sistema no esta en deadlock, aunque exista un ciclo, ya que P3 podrá obtener la instancia de R2 cuando P4 libere la instancia que tiene asignada, rompiendo así el ciclo.
En caso de que en el sistema no exista ciclo entonces de seguro el sistema no esta en deadlock.



Métodos para el manejo de deadlocks

Existen tres métodos diferentes para el manejo de deadlocks:

1. Podemos usar un protocolo para asegurar que el sistema nunca entre en estado de deadlock.
2. Podemos permitir que el sistema entre en estado de deadlock y luego recuperarlo.
3. Podemos ignorar el problema ya que estamos seguros que el sistema nunca entrara en deadlock. Este método es el usado por la mayoría de los sistemas operativos, incluyendo UNIX.

Para asegurar que el deadlock nunca ocurrirá, el sistema puede utilizar la prevención del deadlock, o la anulación del deadlock. La prevención es un conjunto de métodos que aseguran que al menos una de las condiciones necesarias para el deadlock nunca ocurrirá. Estos métodos previenen el deadlocks restringiendo como se puede hacer el pedido de los recursos.
La anulación consiste de que el sistema operativo debe llevar lista de que procesos tiene y pedirán cuales recursos. Con este conocimiento, se puede decidir en caso de que venga un pedido, si le es asignado o el proceso debe esperar. Cada pedido requiere que el sistema considere todos los recursos que hay disponibles, los recursos asignados actualmente a cada proceso, y los futuros pedidos y liberaciones de recursos de los procesos.
Si el sistema no emplea ninguno de los dos métodos, entonces puede ocurrir un deadlock. En este ambiente, el sistema puede proveer un algoritmo que examine el estado del sistema para determinar si se encuentra en deadlock, y un algoritmo para que recupere al sistema del deadlock.
Si el sistema no asegura que el deadlock no ocurrirá, y además no provee mecanismo para la detección y recuperación, entonces puede ocurrir que el sistema entre en estado de deadlock y no se sabe lo que pasa. En este caso, el deadlock no detectado llevara a la caída del rendimiento del sistema, ya que los recursos están siendo mantenidos por procesos que no pueden correr, y cada vez más procesos, por sus sucesivos pedidos, entraran en deadlock. Eventualmente, el sistema parara de funcionar y se necesitara reiniciarlo manualmente.

Aunque éste último método no se ve el más indicado, a veces es utilizado por algunos sistemas operativos, ya que deadlock no ocurre frecuentemente (digamos, una vez por año), por lo que es más barato utilizar éste método en lugar de los demás. También existen circunstancias en donde el sistema se queda congelado sin estar en deadlock. Un ejemplo de esto es: considere un proceso de tiempo real corriendo con una prioridad muy alta (o algún proceso corriendo en algún scheduler que no produce desalojo) y nunca retorna el control al sistema operativo.

Prevención del deadlock

Como se vio, para que un deadlock ocurra se deben cumplir cuatro condiciones. En caso de que evitemos una de ellas estaremos seguros que evitamos la presencia de deadlock. A continuación examinaremos cada una de las 4 condiciones separadamente.

Exclusión mutua: La exclusión mutua debe ser mantenida para recursos del tipo no compartido (impresora). Los procesos compartidos (archivos de solo lectura), por otro lado, no requieren el acceso con exclusión mutua, por lo que no pueden ser involucrados en deadlock. Un proceso nunca necesita esperar para entrar en un recurso compartido. Sin embargo, es muy difícil prevenir deadlock por medio de evitar la condición de exclusión mutua, ya que algunos recursos son sí o sí no compartidos.

Agarrar y esperar: para asegurar que ésta condición nunca ocurra en el sistema, se debe garantizar que cuando un proceso pide un recurso, éste no mantiene ningún otro recurso. Un protocolo que puede ser usado requiere que se le asignen a un proceso todos los recursos antes de que comience su ejecución. Otro protocolo alternativo seria que un proceso puede pedir recursos en caso de que no tenga ningún otro. Si tiene un recurso en su poder y necesita otro, entonces debe liberar el que tiene en su poder.
Para ver la diferencia entre estos dos protocolos, consideremos un proceso que copia datos desde el drive de cinta al archivo en el disco, ordena el archivo, y luego imprime el resultado en la impresora. Si el proceso debe pedir todos los recursos al inicio de su ejecución, entonces debe pedir el drive de cinta, el archivo de disco y la impresora. Así, el proceso mantendrá la impresora durante su entera ejecución aunque solo la necesite al final.
Con el segundo protocolo, se le permite pedir al proceso inicialmente el drive de cinta y el archivo de disco. Este copia desde el drive al disco, luego libera ambos, y pide entonces el archivo de disco nuevamente y la impresora. Luego de copiar el archivo en la impresora, el proceso libera ambos recursos y termina.
Existen dos grandes desventajas de éstos dos protocolos. Primero, la utilización de los recursos puede ser baja, ya que muchos de esos recursos serán asignados pero in-usados por largos periodos. La segunda es que hay peligro de inanición, ya que un proceso que necesita varios recursos importantes puede tener que esperar indefinidamente porque al menos unos de los recursos importantes que necesita esta asignado a otro recurso.

Sin desalojo: La tercera condición necesaria es que no existe desalojo de los recursos que ya han sido asignados. Para asegurar que esta condición no exista, se pueden utilizar el siguiente protocolo. Si un proceso que esta usando algún recurso pide otro que no puede ser asignado inmediatamente (es decir, el proceso debe esperar), entonces los recursos que tenia en su poder le son quitados. Esto es, todos los recursos son implícitamente liberados. Todos aquellos recursos que le son quitados se les agrega a la lista de recursos por los que el proceso esta esperando. El proceso continuara solo si se le pueden asignar los viejos recursos que tenia como así también los nuevos.
Alternativamente, si un proceso pide algunos recursos, primero se chequea si están disponibles. Si lo están, se les asigna. Si no están disponibles, se chequea si los recursos que necesita están asignados a algún otro proceso que está esperando por otros recursos. En caso de que sea así, se le quitan los recursos a los procesos que estaban esperando y se les asignan al proceso que pedía. Por último, en el caso de que los recursos no estén en manos de procesos esperando, entonces el proceso que pide debe esperar. Mientras esta esperando, puede que algunos de sus recursos se le quiten, pero solo por algún proceso que

los pida. Un proceso puede continuar solo si se le asignan los recursos nuevos por los cuales tuvo que esperar y si recupera los recursos que se le quitaron mientras estaba esperando.
Este protocolo se aplica a los recursos cuyo estado se puede registrar fácilmente, tal como registros de CPU y espacio de memoria. No puede, generalmente, ser aplicado a recursos como impresoras o drives de cinta.

Espera circular: Una forma de asegurar que esta condición nunca aparecerá es la de tener un orden total de todos los tipos de recursos, y se requiere que cada proceso pida los recursos en un orden creciente de numeración.
Sea R = {R_1, R_2, R_3, ... R_n} el conjunto de todos los tipos de recursos. Se asigna a cada tipo de recurso un número único, el cual permite hacer la comparación entre diferentes tipos de recursos. Formalmente, se define para cada tipo la función F: R → N, donde N son los números naturales. Por ejemplo, si el conjunto de los tipos de recursos R (el conjunto de todos los tipos) incluye drives de cinta, drives de disco e impresoras, entonces la función F se puede definir como:

F(drive de cinta) = 1;
F(drive de disco) = 5;
F(impresora) = 12;

Ahora si podemos definir el protocolo para prevenir deadlock. Cada proceso puede pedir recursos en un orden creciente de numeración, es decir, un proceso puede pedir inicialmente cualquier cantidad de instancias del tipo R_i. Luego de esto, el proceso puede pedir instancias del tipo de recurso R_j sólo si $F(R_j) > F(R_i)$. En caso de que varias instancias del mismo tipo de recurso sean necesitadas, solo se debe hacer un único pedido para todas las instancias. Por ejemplo, usando el ejemplo anterior, un proceso que quiere pedir el drive de cinta e impresora al mismo tiempo, debe primero pedir el drive de cinta y luego pedir la impresora.
Alternativamente, se puede requerir que cuando un proceso pide una instancia del tipo de recurso R_j, éste tiene que liberar cualquier recurso R_i tal que $F(R_i) \geq F(R_j)$.
Al usar estos dos protocolos, entonces nunca puede ocurrir la condición de espera circular. Note que la función F debe estar definida acorde al normal uso de los recursos del sistema. Por ejemplo, ya que el drive de cinta es usualmente utilizado antes que la impresora, es razonable hacer que F(drive de cinta) < F(impresora).

Anulación del deadlock

El inconveniente que tiene utilizar uno de los 4 métodos que se definieron anteriormente es que baja la utilización de los dispositivos y reduce el throughput del sistema.
Un método adicional para evitar el deadlock es el de utilizar información adicional de cómo serán realizados los pedidos de los recursos. Por ejemplo, si el proceso P pedirá primero el drive de cinta y luego la impresora, y el proceso Q pedirá primero la impresora y luego el drive de cinta, con el conocimiento de cómo serán realizados los pedidos y las liberaciones de los recursos, se puede decidir si, ante un pedido, se hace que el proceso espere o no. Cada pedido requiere que el sistema considere los recursos que están actualmente disponibles, los recursos que están actualmente asignados a los procesos (y a cuales procesos), y los futuros pedidos y liberaciones de cada proceso, para decidir si el actual pedido puede ser satisfecho o debe esperar para evitar un posible deadlock.
El modelo más simple y más usado es el que requiere que cada proceso declare el número máximo de cada tipo de recurso que necesitara. Dando esta información, es posible construir un algoritmo que asegure que el sistema no entrara en deadlock, la idea es que asegura que nunca entrara en espera circular.

Estado seguro: se dice que el sistema esta seguro si puede asignar recursos a cada proceso en algún orden y aun evitar deadlock. Más formalmente, el sistema esta en estado seguro si existe una secuencia segura, es decir, si están los siguientes procesos <P_1, P_2, ..., P_n> para el estado de asignación actual, entonces el sistema esta en estado seguro en caso de que para cada proceso P_i, los recursos que pide pueden ser

satisfechos por los actuales recursos disponibles, más los recursos que son mantenidos por todo aquel proceso P_j, con j < i. En ésta situación, si los recursos que P_i necesita no están disponibles inmediatamente, entonces P_i puede esperar hasta que todo aquel P_j termine. Cuando todos ellos hayan finalizado, P_i puede obtener todos los recursos que necesita, completando así su tarea y liberando los recursos que uso. Cuando P_i termina, entonces P_{i+1} puede obtener los recursos que necesita, y así. En caso de que tal secuencia no exista, entonces se dice que el sistema esta en un estado inseguro.
Habiendo ya dado el concepto de estado seguro, se pueden definir ahora los algoritmos para evitar deadlocks. La idea principal es tratar de que el sistema siempre permanezca en estado seguro. Inicialmente el sistema esta en estado seguro. Cuando un proceso pide un recurso que esta disponible, el sistema debe decidir si se le otorga el recurso inmediatamente o el proceso deba esperar. El pedido es concedido en caso de que tal asignación deje al sistema en un estado seguro.
Note que en este sistema, si un proceso pide un recurso que esta disponible, igualmente cabe la posibilidad de que deba esperar, por lo que la utilización de recursos puede ser más baja que si el sistema no tuviera tales algoritmos de prevención.

Algoritmo de grafo de asignación de recursos: En caso de tener un sistema de asignación de recursos con solo una instancia de cada tipo de recurso, entonces se puede aplicar otro método de asignación en lugar del método de grafo anteriormente descripto.
Además de los arcos de pedidos y asignaciones, ahora introducimos otro tipo de arco llamado arco de demanda, el cual si tenemos un arco de demanda entre el proceso P_i y el recurso R_j, entonces el proceso P_i puede pedir el recurso R_j en algún momento determinado del futuro. Estos arcos se parecen a los arcos de pedido en dirección, solo que son representados por líneas punteadas. Cuando el proceso P_i pide el recurso R_j, el arco de demanda se convierte en un arco de pedido. Similarmente, cuando un recurso R_j es liberado por P_i, el arco de asignación se convierte ahora en un arco de demanda. Notar que ahora la demanda de los recursos debe ser hecha a priori, es decir, antes de que un proceso comience debe hacer la demanda de todos los recursos que necesitara apareciendo en el grafo todos sus arcos de demanda.
Supongamos que el proceso P_i pide el recurso R_j. El pedido se puede conceder si convirtiendo el arco de pedido $P_i \rightarrow R_j$ a un arco de asignación $R_j \rightarrow P_i$ no resulta en la formación de un ciclo en el grafo. Note que el chequeo de si el sistema esta en estado seguro se hace por medio de un algoritmo de detección de ciclos en un grafo. Si no existen ciclos, la asignación del recurso dejara al sistema en estado seguro. De otra manera, el proceso P_i deberá esperar.

Algoritmo del Banquero: un algoritmo de asignación de recursos con grafo no es aplicable cuando existen varias instancias de un tipo de recurso. El siguiente algoritmo si es aplicable a tales sistemas pero es menos eficiente. Este algoritmo es conocido como algoritmo del banquero. Cuando un nuevo proceso entra al sistema, éste debe declarar el máximo número de instancias de cada tipo de recursos que necesitara. Este número no debe exceder al número total de recursos del sistema. Cuando un usuario pide un conjunto de recursos, el sistema debe analizar si al asignarle tales recursos el sistema sigue en estado seguro. Si es así, se le asignan los recursos, en caso contrario, el proceso debe esperar hasta que algún otro proceso libere los recursos que necesita.
Varias estructuras se necesitan para poder llevar a cabo dicho algoritmo. Si n es el número de procesos que tiene el sistema y m el número de tipos de recursos que tiene el sistema, entonces se necesitan las siguientes estructuras:

- *Available:* es un vector de largo m que indica el número de recursos disponibles de cada tipo. Si available[j] = k, entonces hay k instancias del tipo de recurso R_j disponibles.
- *Max:* es una matriz de n * m que define la demanda máxima de cada recurso. Si max[i,j] = k, entonces P_i puede pedir a lo sumo k instancias del tipo de recurso R_j.
- *Allocation:* es una matriz de n * m que define el número de recursos de cada tipo asignados actualmente a cada proceso. Si allocation[i,j] = k, entonces el proceso P_i tiene asignado actualmente k instancias del recurso de tipo R_j.

- *Need:* es una matriz de n * m que indica el número de recursos restantes de cada tipo necesitados para cada proceso. Si need[i,j] = k, entonces P_i puede necesitar k instancias más del tipo de recurso R_j para completar su tarea. Note que need[i,j] = max[i,j] – allocation[i,j].

Estas estructuras de datos varían con el tiempo en espacio y valor.
Para simplificar la notación, trataremos a cada fila de las matrices need y allocation como vectores y nos referiremos a ellos como *allocation*$_i$ y *need*$_i$. Ante esto, el vector allocation$_i$ especifica los recursos actualmente asignados al proceso P_i.

Algoritmo de seguridad: El algoritmo que descubre si el sistema esta o no en un estado seguro es el siguiente:

1. Sea work y finish dos vectores de largo m y n respectivamente. Inicializar work := available y finish := false para cada elemento de finish.
2. Encontrar un i que cumpla éstas dos condiciones:
   - finish[i] = false;
   - need$_i$ <= work

   En caso de que tal i no exista ir al paso 4.
3. work := work + allocation$_i$
   finish[i] := true
   ir al paso 2
4. Si finish[i] = true para todos los i, entonces el sistema esta en estado seguro.

Algoritmo de asignación de recursos: Sea request$_i$ el vector de pedidos del proceso P_i. Si request$_i$[j] = k, entonces el proceso P_i quiere k instancias del tipo de recurso R_j. Cuando el proceso P_i hace el pedido de sus recursos, se deben tomar las siguientes acciones:

1. Si request$_i$ <= need$_i$, ir al paso 2. De otra forma, elevar una condición de error, ya que el proceso ha excedido su demanda máxima.
2. Si request$_i$ <= available, ir al paso 3. De otra forma, P_i debe esperar ya que los recursos no están disponibles.
3. El sistema pretende asignarle los recursos pedidos por el proceso P_i, por lo que se deben llevar a cabo las siguientes modificaciones:
   - available := available – request$_i$;
   - allocation$_i$:= allocation$_i$ + request;
   - need$_i$:= need$_i$ – request$_i$;

Si el resultado de la asignación de recursos resulta en un estado seguro, la transición es completada y al proceso P_i se le asignan los recursos. Sin embargo, si el nuevo estado no es seguro, entonces P_i debe esperar por request$_i$ y se restaura el viejo estado de asignación de recursos.

Detección del deadlock

En caso de que el sistema no provea mecanismos para la prevención de deadlock ni para la anulación de deadlock, entonces, y ante el hecho de que el deadlock pueda ocurrir, debe proveer:

- un algoritmo que examine el estado del sistema para determinar si está o no en deadlock.
- un algoritmo para recuperar al sistema del deadlock.

A continuación se verán dos métodos, uno que se utiliza cuando existe una instancia de cada tipo de recurso y el siguiente cuando existen varias instancias. Se debe tener en cuenta que los esquemas de detección y recuperación requieren un overhead que no incluye solo el costo del tiempo de corrida de mantener la información necesaria y la ejecución del algoritmo de detección, sino también las pérdidas con respecto a la recuperación.

Una única instancia para cada tipo de recurso: en caso de que todos los tipos de recursos tengan solo una instancia, entones podemos definir un método de detección y recuperación de deadlock llamado grafo *espera-por*. Este grafo se obtiene a partir del grafo de asignación de recursos eliminando los nodos de tipo de recursos y eliminando los arcos adecuados.
Un arco desde el proceso P_i a P_j en el grafo *espera-por* implica que el proceso P_i esta esperando para que el proceso P_j libere un recurso que P_i necesita. Así, un arco en el grafo *espera-por* existe entre $P_i \rightarrow P_j$ si y sólo si en el correspondiente grafo de asignación de recursos existen los arcos $P_i \rightarrow R_k$, y $R_k \rightarrow P_j$. En la figura 7.7 se ve el grafo de asignación de recursos y el grafo de espera correspondiente.

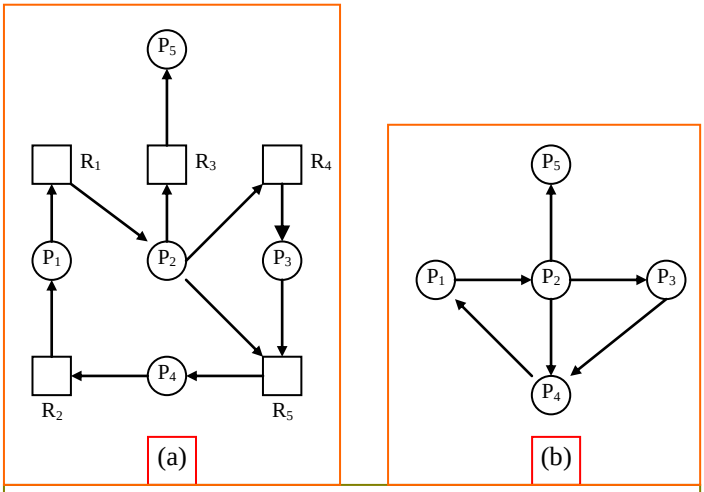


Figura 17.1 (a) Grafo de asignación de recursos. (b) Correspondiente grafo de espera.

Como antes, existe deadlock solo si el grafo *espera-por* contiene un ciclo. Para detectar deadlock, el sistema necesita mantener un grafo *espera-por* y periódicamente invocar a un algoritmo de detección de ciclo en un grafo.

Varias instancias de un tipo de recurso: a continuación se explica un algoritmo que detecta deadlock si el sistema tiene varias instancias de un mismo tipo recurso. Este algoritmo emplea varias estructuras de datos que varían con el tiempo similares a las del algoritmo del banquero.

- *available:* es un vector de largo m que indica el número de recursos disponibles para cada tipo.
- *allocation:* es una matriz de n * m que define el número de recursos de cada tipo que están actualmente asignados a cada proceso.
- *request:* es una matriz de n * m que indica el actual pedido de cada proceso. Si request[i,j] = k, entonces el proceso P_i esta pidiendo k instancias más del tipo de recurso R_j.

El algoritmo de detección descripto simplemente investiga cada asignación que se hace para los diferentes procesos.

1. Sea work y finish vectores de largo m y n respectivamente. Inicializar work := available. Para cada i entre 1 y n, si $allocation_i \neq 0$, entonces finish[i] := false; de otra forma, finish[i] := true.
2. Encontrar un i que cumpla las condiciones:
   - finish[i] = false;
   - $request_i <= work$.

   En caso de que tal i no exista, ir al paso 4.
3. work := work + $allocation_i$
   finish[i] := true
   ir al paso 2.

4. Si finish[i] = false para algún i entre 1 y n, entonces el sistema esta en deadlock. Más precisamente, si finish[i] = false, entonces el proceso P_i esta en deadlock.

Recuperación del deadlock

Cuando un algoritmo de detección de deadlock descubre que existe tal, existen varias alternativas para tomar. Una posibilidad seria informar al operador que el sistema esta en deadlock y que el operador trate el deadlock manualmente. La otra posibilidad es que el sistema recupere el estado seguro automáticamente. Existen dos opciones para romper el deadlock. Una posibilidad seria simplemente abortar uno o más procesos para romper la espera circular. La segunda opción es la de desalojar alguno de los recursos de los procesos que están en deadlock.

Terminación de procesos: Para eliminar el deadlock por medio de la suspensión de la ejecución de procesos existen dos métodos. En ambos métodos el sistema reclama todos los recursos asignados a los procesos que serán terminados.

- Abortar todos los procesos en deadlock.
- Abortar de un proceso a la vez hasta que el ciclo de deadlock este eliminado.

Se debe tener en cuenta que el abortar un proceso puede no ser fácil. En caso de que el proceso este en medio de una modificación de un archivo, terminar el proceso provocaría dejar el archivo en un estado incorrecto. Similarmente, si el proceso esta en el medio de una impresión, el sistema debe resetear el estado de la impresora a un estado correcto para así puede imprimir correctamente el siguiente trabajo.
En caso de que se use el método de terminar algunos procesos hasta que se rompa el deadlock, se debe determinar cuales de los procesos que están en deadlock son los que se terminaran. Esta decisión es similar al problema de CPU scheduling. Básicamente la cuestión es económica, se deben terminar aquellos procesos cuya terminación provoque costos bajos.

III. Administración del almacenamiento

El propósito más importante de un sistema es el de ejecutar programas. Estos programas, juntos con los datos que ellos acceden, deben estar en la memoria principal (al menos parcialmente) durante la ejecución.
Para mejorar tanto la utilización de la CPU como la velocidad con que ésta responde a los usuarios, la computadora debe mantener varios procesos en memoria. Existen muchos esquemas de administración de memoria diferentes. La elección de cual es el mejor depende principalmente del hardware, debido que cada algoritmo requiere su propio soporte del hardware.
Ya que la memoria principal es demasiada chica para mantener todos los datos y programas permanentemente, el sistema debe proveer almacenamiento secundario (tal como discos) que trabaje como copia de la memoria.
La mayor parte de las computadoras tienen una jerarquía de memoria, con una cantidad pequeña de memoria caché muy rápida, costosa y volátil; algunos megabytes de memoria principal (RAM) volátil de mediana velocidad y mediano precio; cientos o miles de megabytes de almacenamiento en disco lento, económico y no volátil. Corresponde al sistema operativo coordinar el uso de éstas memorias.

8. Administración de la memoria

Para permitir que la CPU sea compartida por varios procesos, se deben mantener en memoria varios procesos, por lo que debemos compartir memoria. En este capítulo, veremos varios mecanismos para administrar la memoria.

Vistazo

La memoria consta de un largo arreglo de palabras o bytes, cada una con su propia dirección. La CPU saca las instrucciones desde la memoria según la dirección que se encuentra en el contador del programa. Éstas instrucciones pueden provocar cargas de, y almacenamiento a específicas direcciones de memoria.
Un ciclo de ejecución de instrucción común primero obtiene una instrucción de la memoria. La instrucción se decodifica y puede causar la búsqueda de operandos a la memoria. Luego de que la instrucción ha sido ejecutada sobre los operandos, el resultado puede que deba ser almacenado otra vez en la memoria. Note que la unidad de memoria ve solo un string de direcciones de memoria; ella no conoce como son generados (el contador de instrucción, la indexación, la indirección, direcciones propiamente dichas, etc.), o qué son (instrucciones o datos). Acordemente, podemos ignorar como es generada una dirección de memoria por el programa que está corriendo.

Address binding: usualmente un programa reside en disco como un archivo binario ejecutable. El programa debe ser traído a la memoria y ubicado dentro de un proceso para que sea ejecutado. Dependiendo del modelo de administración de memoria, puede que el proceso durante su ejecución sea movido entre el disco y la memoria. La colección de procesos en disco que están esperando para ser puestos en memoria y así poder ser ejecutados forman la cola de entrada.
El procedimiento normal es seleccionar uno de estos procesos de la cola de entrada y cargarlo en la memoria. Al terminar, el espacio de memoria que éste ocupaba se declara disponible.
La mayoría de los sistemas permiten que los proceso del usuario estén en cualquier lugar de la memoria. Así, aunque el espacio de direcciones de la maquina empiece en 00000, la primera dirección del proceso del usuario no necesita estar en la dirección 00000. Ante esto, el direccionamiento que el programa del usuario utiliza es afectado. En la mayoría de los casos, un programa de usuario atraviesa varios pasos antes que pueda ser ejecutado (figura 8.1). Las direcciones pueden estar representadas de diferentes formas en los diferentes pasos. Las direcciones en el programa fuente son generalmente simbólicas (tal como COUNT). Un compilador típicamente ligara estas direcciones simbólicas a direcciones relocalizables (tal como "14 bytes desde el comienzo del módulo"). El editor de linkeo o cargador ligara éstas direcciones

relocalizables a direcciones absolutas (tal como 74014). Cada ligamiento es un mapeo desde un espacio de direcciones a otro.
Clásicamente, el ligamiento de instrucciones y datos a direcciones de memoria puede ser echo en cualquiera de los siguientes pasos:

- *Tiempo de compilación:* si es conocido en tiempo de compilación donde el proceso residirá en memoria, entonces se puede generar *código absoluto*. Por ejemplo, si se conoce a priori que un proceso de usuario residirá en memoria comenzando de la dirección R, entonces el código generado por el compilador comenzara en dicha dirección y se extenderá desde allí. Si, un tiempo más tarde, la dirección de comienzo cambia, entonces será necesario recompilar el código. Los programas de formatos .COM de MS-DOS son de código absoluto limitado a tiempo de compilación.
- *Tiempo de carga:* si no se conoce en tiempo de compilación donde residirá el proceso en memoria, entonces el compilador debe generar código relocalizable. En este caso, el ligamiento final es demorado hasta el tiempo de carga. En caso de que la dirección de comienzo cambie, solo se necesita recargar el código del usuario para reflejar éste cambio.
- *Tiempo de ejecución:* en caso de que el proceso pueda ser movido durante su ejecución desde un segmento de memoria a otro, entonces el ligamiento debe ser demorado hasta el tiempo de ejecución. Se necesita un hardware especial para este esquema de trabajo.

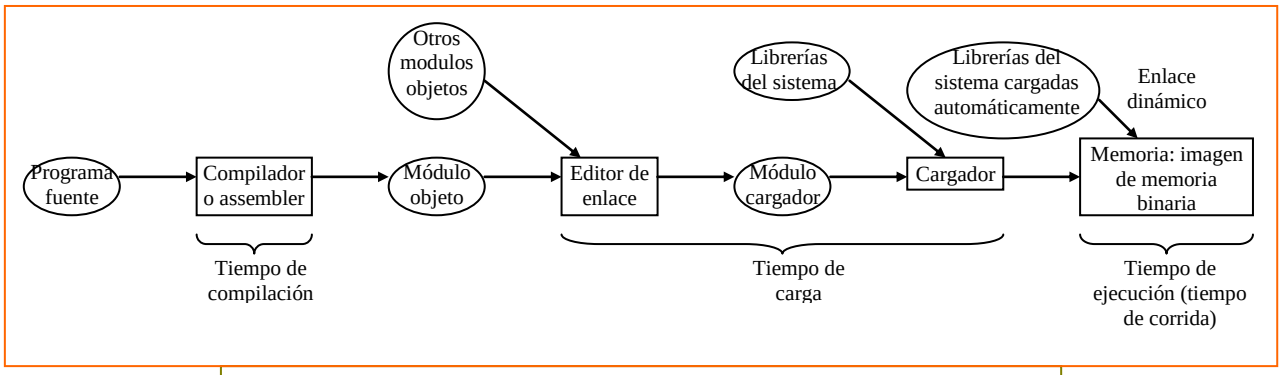


Figura 8.1 Procesamiento multi-etapa de un proceso de usuario.

Carga dinámica: El programa entero y los datos de un proceso deben estar en la memoria física para ejecutar el proceso. El tamaño de un proceso es limitado al tamaño de la memoria física. Para obtener mejor utilización del espacio de la memoria se utiliza la carga dinámica, con lo que una rutina no es cargada hasta que no es llamada. Todas las rutinas se mantienen en disco con un formato de carga relocalizable. El programa principal (main) es cargado en la memoria y ejecutado. Cuando una rutina necesita llamara a otra rutina, la rutina que llama primero chequea para ver si la rutina que necesita ha sido cargada, es decir, está en la memoria. En caso de que no este, se llama al cargador de enlace relocalizable para que cargue la rutina deseada en memoria, y para cambiar la tabla de direcciones del programa para reflejar éste cambio. Luego, el control es pasado a la nueva rutina cargada.
La ventaja de la carga dinámica es que una rutina que no es usada nunca es cargada. Este sistema es muy usado cuando se necesita grandes cantidades de código para manejar casos que ocurren infrecuentemente, tal como las rutinas de error. Con este esquema, aunque el tamaño total del programa pueda ser muy grande, la porción que esta siendo actualmente usada (y por lo tanto la porción que esta cargada) puede ser muy pequeña.
La carga dinámica no requiere soporte del sistema operativo, ya que es responsabilidad de los usuarios diseñar programas que tengan la ventaja de tal esquema. Sin embargo, el sistema operativo puede ayudar a los programadores proveyendo rutinas en forma de librerías que implementan la carga dinámica.

Enlace dinámico: Note que la figura 8.1 también muestra librerías de enlace dinámico. La mayoría de los sistemas operativos soportan el enlace estático, en el cual las librerías del lenguaje del sistema son tratadas como cualquier otro módulo objeto y son combinadas por el cargador en la imagen del programa

binario. El concepto de enlace dinámico es similar al de carga dinámica. En lugar de que la carga se posponga hasta el tiempo de ejecución, ahora el enlace es pospuesto hasta el tiempo de ejecución. Esta característica es usualmente usada con las librerías del sistema, tal como las librerías de subrutinas de un determinado lenguaje. Sin ésta facilidad, todos los programas en el sistema necesitarían tener una copia de sus librerías del lenguaje (o al menos las que el programa necesita) incluidas en la imagen ejecutable. Este requerimiento gasta tanto espacio en el disco como en la memoria. Con el enlace dinámico, un stub se incluye en la imagen de cada referencia a una rutina de librería. Este stub es una pequeña pieza de código que indica como ubicar la rutina de librería apropiada en la memoria, o como cargar la librería si la rutina no esta aun presente.
Al ejecutarse el stub, éste chequea para ver si la rutina ya está en memoria. En caso de que no este, el programa la carga en memoria. De cualquier modo, el stub se reemplaza con la dirección de la rutina, y ejecuta la rutina. Así, la siguiente vez que el segmento de código es necesitado, la rutina de librería es ejecutada directamente, sin incurrir costo en el enlace dinámico. Bajo este esquema, todos los procesos que utilizan una librería de un lenguaje ejecutan solo una copia del código de librería.
Esta característica puede ser extendida a las modificaciones de librería. Una librería puede ser reemplazada por una versión más nueva, y todos los programas que hacen referencia a dicha librería automáticamente usaran la nueva versión. Sin el enlace dinámico, todo éstos programas necesitaran ser relinkeados para ganar el acceso a la nueva librería. Para que los programas no ejecuten accidentalmente versiones incompatibles de librería, se incluye en los programas y en las librerías información de versión. Más de una versión de librería pueden ser cargadas en la memoria, y cada programa usara su información de versión para decidir que versión de librería utilizara. Los cambios de poca importancia retienen el mismo número de versión, mientras que grandes cambios incrementan el número de versión. Así, solo los programas que son compilados con la nueva versión de librería son afectados por la incompatibilidad de los cambios incorporados en ésta. Otros programas enlazados antes de que la nueva versión de librería sea instalada continuaran utilizando la vieja versión. Este sistema es también conocido como librerías compartidas.
No como la carga dinámica, el enlace dinámico generalmente requiere alguna ayuda del sistema operativo. Si los procesos en memoria están protegidos entre sí, entonces el sistema operativo es la única entidad que puede chequear para ver si la rutina necesitada está en el espacio de direcciones de otro proceso y puede permitir que múltiples procesos accedan a la misma dirección de memoria.

Overlays: Para que un proceso pueda ser más grande que la cantidad de memoria que se le asigna, se utiliza una técnica llamada overlays. La idea es la de mantener en memoria aquellas instrucciones y datos que son necesitados. Cuando se necesiten otras instrucciones, éstas son cargadas en el espacio de direcciones que fueron ocupadas por las instrucciones anteriores que ahora no se necesitan.
Como ejemplo, consideremos un assembler de dos pasos. En el paso 1, el assembler construye una tabla de símbolos; en el paso 2, genera el código en lenguaje de maquina. Podemos particionar todo en el código del asembler en el código del paso 1, el código del paso 2, la tabla de símbolos que se forma y rutinas comunes del paso 1 y 2:

Paso 1:	70K
Paso 2:	80K
Tabla de símbolos:	20K
Rutinas comunes:	30K

Para cargar todo en una sola vez, requeriríamos 200K de memoria. Si solo hay disponible 150K, nuestro proceso no se podría correr. Sin embargo, note que el paso 1 y 2 no necesitan estar en memoria al mismo tiempo. Ante esto, definimos dos overlays: el overlay A es la tabla de símbolos, las rutinas comunes y el paso 1; mientras que el overlay B es la tabla de símbolos, las rutinas comunes y el paso 2. Se ha agregado un driver de overlay (10K) y comienza con el overlay A en memoria. Al finalizar el paso 1, se salta al driver del overlay, el cual lee el overlay B de la memoria, sobrescribiendo el overlay A, y luego transfiere el control al paso 2 (Figura 8.2). Ante esto, se puede correr el programa con solo 150K. Sin embargo, correrá algo más lento debido a la I/O extra para leer el código del overlays B.

Los códigos del overlay A y el del B son mantenidos en disco como imágenes de memoria absoluta, y son leídas cuando el driver del overlay las necesite.

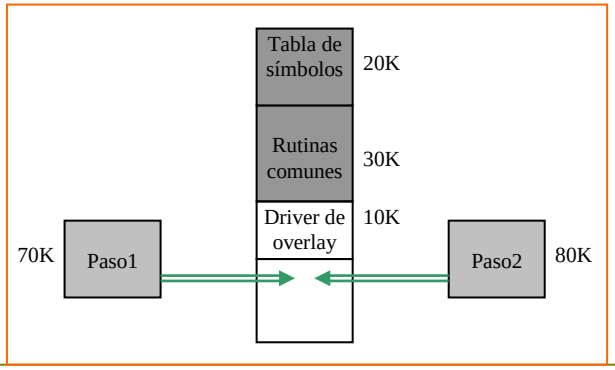


Figura 8.2 Overlays de un asembler de dos pasos.

Como en la carga dinámica, overlay no necesita soporte especial del sistema operativo, ya que puede ser implementada por el usuario con simples estructuras de archivo, leyendo desde el archivo a la memoria y luego saltando a la memoria y ejecutando las recientes instrucciones leídas.

Espacio de direcciones lógicas versus físicas

Una dirección generada por la CPU se dice que es una dirección lógica, mientras que una dirección vista por la unidad de memoria (es decir, una cargada en el *registro de dirección de memoria* de la memoria) se dice que es una dirección física.

Los esquemas de address-binding en tiempo de compilación y en tiempo de carga resultan en un ambiente donde las direcciones lógicas y físicas son las mismas. Sin embargo, el esquema de address-binding en tiempo de ejecución producen un ambiente donde las direcciones lógicas y físicas son diferentes. En este caso, usualmente nos referimos a una dirección lógica como una dirección virtual. Al conjunto de todas las direcciones lógicas generadas por un programa es llamado el espacio de direcciones lógicas; el conjunto de direcciones físicas correspondientes a éstas direcciones lógicas es referenciado como el espacio de direcciones físicas.

El mapeo en tiempo de ejecución desde la direcciones lógicas a las direcciones físicas es hecho por la unidad de administración de memoria (memory-management unit: MMU), el cual es un dispositivo de hardware. Existen varios esquemas para lograr este mapeo que se verán más adelante.

En la figura 8.3 se ve un ejemplo. El registro base es llamado registro de reubicación. El valor del registro de reubicación es sumado a cada dirección generada por el proceso de usuario en el momento que ésta es enviada a memoria.

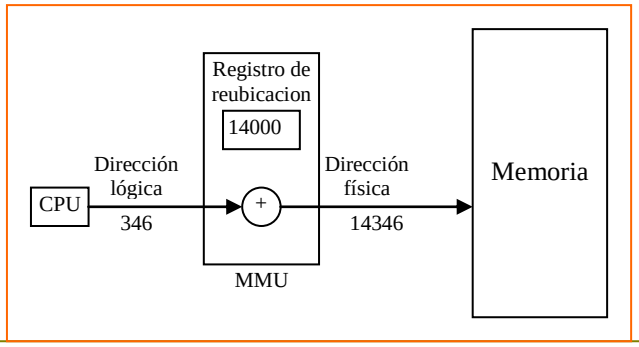


Figura 8.3 Reubicación dinámica usando un registro de reubicación.

Note que el programa del usuario nunca ve las direcciones físicas reales. El programa puede crear un puntero a la ubicación 346, almacenarlo en memoria, manipularlo, compararlo con otras direcciones (todas estas operaciones como el número 346). Solo cuando éste (es decir, el 346) es usado como una dirección

de memoria (quizás en una carga o almacenamiento indirecto), ésta es reubicada de acuerdo al registro base. El programa del usuario trata con direcciones lógicas. El hardware de mapeo de memoria convierte las direcciones lógicas a físicas. La ubicación final de una dirección de memoria referenciada no es determinada hasta que se hace la referencia.
Note también que tenemos dos tipos diferentes de direcciones: direcciones lógicas (en el rango 0 a *max*) y direcciones físicas (en el rango R + 0 a R + *max* para el valor base R). El usuario genera solo direcciones lógicas y piensa que el programa corre en las ubicaciones 0 a *max*.

Swapping

Un proceso debe estar en memoria para poder ser ejecutado. Sin embargo, un proceso puede ser sacado temporalmente de la memoria a un almacenamiento de seguridad, y luego traído nuevamente a la memoria para continuar su ejecución. Por ejemplo, supongamos un ambiente multiprogramado con un algoritmo de scheduling de CPU Round Robin. Cuando un quantum termina, el administrador de memoria comenzara a sacar de la memoria el proceso que finalizo, y traerá otro proceso al espacio de memoria que ha sido liberado (Figura 8.4). Mientras tanto, el scheduler de CPU asignara una porción de tiempo de CPU a otro proceso que este en la memoria. Cada vez que un proceso termina su quantum, será cambiado con otro proceso.

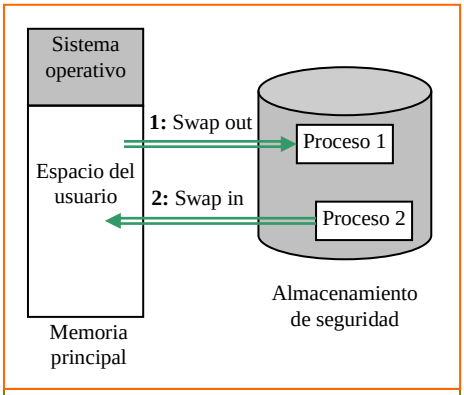


Figura 8.4 Cambio de dos procesos.

Una variante de esta política de quantum es usar algoritmos de scheduling basado en prioridades. Si un proceso de una alta prioridad llega y quiere servicio, el administrador de memoria puede sacar un proceso de más baja prioridad para cargar el de alta prioridad y ejecutarlo. Al terminar, el proceso de baja prioridad puede ser traído nuevamente y continuar. Esta variante es llamada *roll in, roll out*.
Normalmente, un proceso que es sacado de la memoria será retornado a la misma en el mismo espacio de direcciones que ocupaba antes de que sea desalojado. Esta restricción depende del método de address-binding que se haya implementado. Si el binding es hecho en tiempo de carga o de compilación, entonces el proceso debe mantener su espacio de direcciones. Si el binding fue realizado en tiempo de ejecución, entonces es posible cambiar un proceso en diferentes espacios de direcciones, ya que las direcciones físicas son calculadas en tiempo de ejecución.
Swapping requiere un almacenamiento de seguridad. Este es comúnmente un disco rápido. Debe ser bastante grande para mantener las copias de todas las imágenes de memoria para todos los usuarios, y debe proveer acceso directo a éstas imágenes de memoria. El sistema mantiene una cola de listos el cual son todos los procesos cuyas imágenes de memoria están almacenadas ya sea en la memoria o en disco y están listos para correr. Cuando el scheduler de la CPU decide ejecutar un proceso, éste llama al dispatcher. El dispatcher chequea para ver si el siguiente proceso esta en memoria. Si no esta, y no hay memoria libre, el dispatcher saca (swap out) un proceso que esta en memoria y pone (swap in) el proceso deseado. Luego recarga sus registros y transfiere el control al proceso seleccionado. Se debe tener claro que el tiempo de context switch en estos sistemas es bastante alto.

Existen otras restricciones sobre swapping. Si queremos cambiar un proceso, debemos estar seguros que está completamente ocioso. Una preocupación particular es cualquier I/O pendiente. Si un proceso esta esperando por una operación de I/O, puede que queramos cambiar el proceso y así poder liberar su memoria. Sin embargo, si la I/O esta accediendo asincrónicamente a los buffer de I/O de la memoria del usuario (es decir, el dispositivo de I/O esta usando parte del espacio de memoria del proceso a sacar), entonces el proceso no puede ser cambiado. Asumamos que la operación de I/O esta en una cola de espera ya que el dispositivo esta ocupado. Entonces, si sacamos el proceso P_1 y metemos el proceso P_2, la operación de I/O podría entonces intentar usar la memoria que ahora pertenece al proceso P_2. Las dos soluciones principales son:

- nunca cambiar un proceso que tiene una I/O pendiente.
- ejecutar las operaciones de I/O solo en buffers del sistema operativo (que trabajan en la parte de memoria del sistema operativo, el cual nunca es tocada por los procesos del usuario). La transferencia de memoria entre el sistema operativo y el proceso ocurrirá solo cuando el proceso este cargado.

Asignación continua

La memoria principal debe alojar tanto al sistema operativo como a los procesos del usuario. La memoria se divide en dos partes, una donde reside el sistema operativo y la otra para los procesos del usuario. El sistema operativo puede estar tanto en la parte alta como en la baja de la memoria aunque, donde el más afectado de esta decisión es el vector de interrupciones. Debido a que el vector de interrupciones se aloja generalmente en la parte baja, el sistema operativo también se ubica en la parte baja (Figura 8.5).

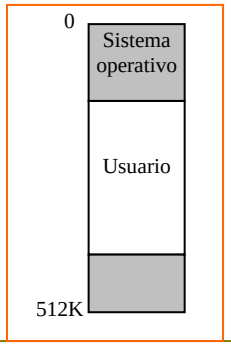


Figura 8.5 Partición de la memoria

Asignación de partición única: Si el sistema operativo esta en la parte baja de la memoria y los procesos del usuario en la parte alta, se necesita proteger al código y los datos del sistema operativo de cambios que puedan producir los procesos del usuario. Además se necesita proteger los procesos del usuario entre sí. Se puede proveer ésta protección por medio del registro de reubicación que se explico anteriormente, más un registro limite. El registro de reubicación contiene el valor de la dirección física más pequeña; el registro limite contiene el rango de direcciones lógicas(por ejemplo, reubicación = 100.040, y límite = 74.600). Con ambos registros, cada dirección lógica debe ser menor que el registro limite; la MMU mapea la dirección lógica dinámicamente sumando el valor del registro de reubicación. Esta dirección mapeada es enviada a la memoria (Figura 8.6).

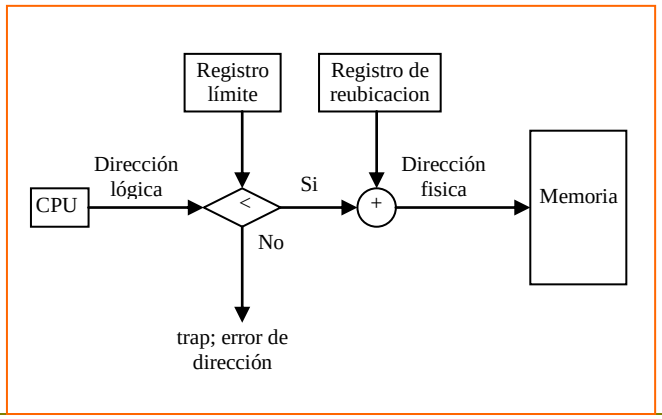


Figura 8.6 Soporte de hardware para los registros límite y de reubicación

Cuando el scheduler de la CPU selecciona un proceso para ejecutarlo, el dispatcher carga ambos registros con los valores correctos como parte del context switch. Ya que cada dirección generada por la CPU es chequeada contra estos registros, podemos proteger tanto al sistema operativo como a programas y datos de otros usuarios de posibles modificaciones que pueda hacer éste proceso corriendo.
Note que éste esquema provee una forma para permitir que el tamaño del sistema operativo cambie dinámicamente. Esto es deseable en muchos casos. Por ejemplo, el sistema operativo contiene espacio de código y para buffers para los drivers de los dispositivos. Si un driver de dispositivo (u otro servicio del sistema operativo) no es comúnmente usado, no es deseable mantener su código y datos en memoria, ya que se podría usar dicho espacio para otros propósitos. Tal parte de código del sistema operativo es llamado migratorio (*transient*).

Asignación de partición múltiple: un problema se produce en el momento que existe un lugar de memoria libre y hay varios procesos en la cola de listos para entrar en la memoria, es decir, si hay un lugar de memoria, como lo dividimos entre todos los procesos esperando. Un esquema simple es asignación de memoria es dividir la memoria en un número de particiones de tamaño fijo. Cada partición puede contener solo un proceso. Así, el grado de multiprogramación es limitado por el número de particiones. Cuando una partición esta libre, se selecciona un proceso de la cola de listos y se carga en la partición de memoria libre. Al terminar el proceso, la partición esta nuevamente libre para otro proceso. El sistema operativo mantiene una tabla donde se indica que partes de la memoria están disponibles y cuales están ocupadas. Inicialmente, la memoria esta disponible para los procesos del usuario, y es considerada como un gran bloque de memoria disponible, un hole. Cuando un proceso llega y necesita memoria, se busca un agujero bastante grande para poder guardar dicho proceso. En caso de encontrar uno, se asigna tanta memoria como el proceso necesite, manteniendo el resto disponible para otros procesos.
Por ejemplo, asumamos que tenemos una memoria disponible de 2560K y el sistema operativo reside en 400K. Ante esto tenemos 2160K para los procesos del usuario. En la figura 8.7 se ve la cola de entrada, y un scheduling de trabajos FCFS, por lo que se puede asignar inmediatamente memoria a los procesos P_1, P_2, y P_3 (figura a). Luego solo se tiene un agujero de 260K el cual no puede ser usado por ninguno de los procesos restantes en la cola de entrada. Usando un scheduling de CPU Round Robin con un quantum de una unidad de tiempo, el proceso P_2 terminara a las 14 unidades, liberando su memoria (figura b). De la cola de trabajo el scheduler toma el siguiente proceso, es decir P_4 (figura c). El proceso P_1 terminara a las 28 unidades (figura d), produciendo que el scheduler tome a P_5 (figura e).

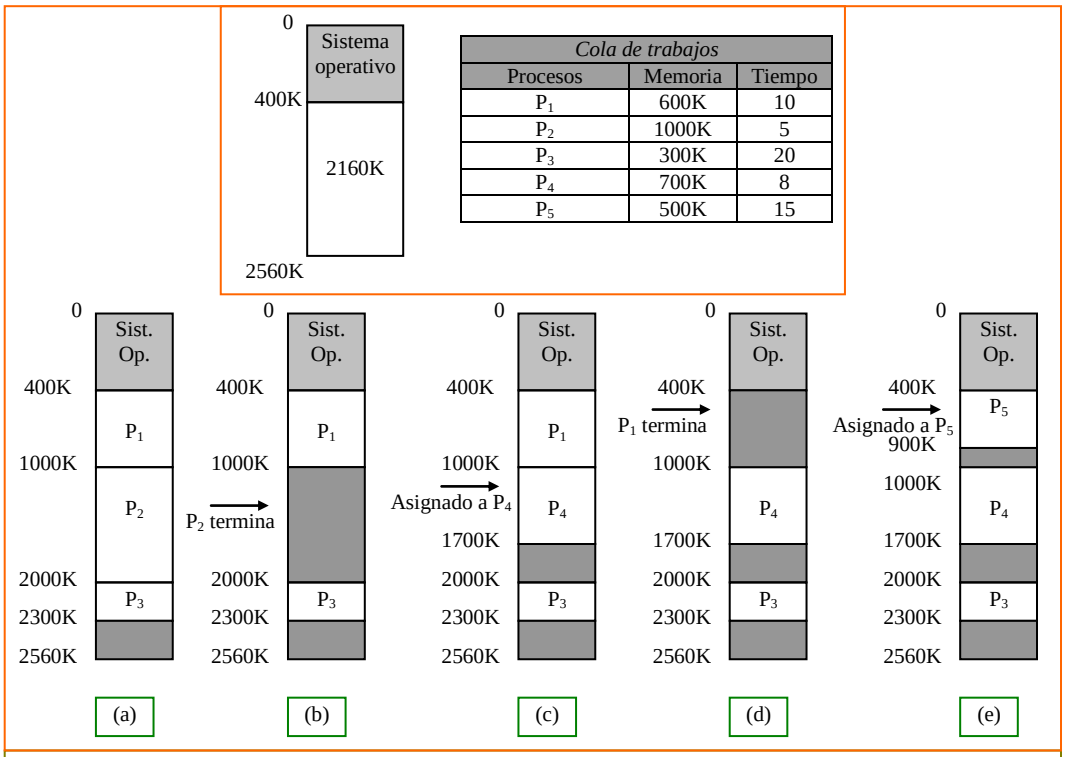


Figura 8.7 Asignación de memoria en un scheduling de largo plazo.

En general, en cualquier momento se tiene una lista de agujeros dispersos en la memoria, el cual difieren en su tamaño, y una cola de entrada de procesos. Al llegar un proceso y necesitar memoria, se busca del conjunto de agujeros si existe uno bastante grande como el proceso. En caso de que el agujero sea más grande, se divide en dos, una parte es la parte que se le asigna al proceso y la otra se agrega a la lista de agujeros. Al terminar un proceso, éste libera su memoria, por lo que dicho bloque libre se ingresa a la lista de agujeros. Si existen dos agujeros adyacentes, pueden ser unidos para formar un único agujero grande. En este punto, se puede recorrer la cola de listos para ver si existe algún proceso que entre en alguno de los agujeros.

Este problema de asignación de memoria entre procesos se conoce como problema de asignación de almacenamiento dinámico, es decir, dado un pedido de memoria de un proceso de tamaño n, que agujero asignarle. Hay muchas soluciones a éste problema. La lista de agujeros es recorrida para ver cual es el mejor agujero para el proceso. Las estrategias son:

- First-fit: asignar el primer agujero que sea suficientemente grande.
- Best-fit: asignar el agujero más chico que sea suficientemente grande. En esta estrategia se debe recorrer la lista completa, a menos que la lista este ordenada por tamaños.
- Worst-fit: asignar el agujero más grande. Una vez más, se debe recorrer la lista completa, a menos que este ordenada por tamaño.

Simulaciones han demostrado que worst-fit es la peor estrategia. El más rápido de los restantes es first-fit.

Fragmentación externa e interna: el algoritmo anterior sufre de fragmentación externa. Los procesos son cargados y eliminados de la memoria, por lo que la memoria libre se va rompiendo en pequeñas piezas. La fragmentación externa existe cuando el espacio total de memoria libre es suficiente para satisfacer un pedido, pero ésta no es continúa, ya que se tienen varios pequeños agujeros. La selección de first-fit versus best-fit puede afectar la cantidad de fragmentación (first-fit es mejor para algunos sistemas, mientras que berst-fit es mejor para otros).

Otro problema que surge con el esquema de asignación de múltiple partición es el siguiente: supongamos que tenemos un agujero de 18646 bytes y el siguiente proceso pide 18642 bytes. Si asignamos al proceso la memoria exacta que pide nos quedaría un agujero de 2 bytes. El overhead por mantener información de dicho agujero es mucho más grande que el agujero por sí mismo. La solución es la de asignar al proceso un poco más de memoria que la que pide. La diferencia entre éstos dos números es la fragmentación interna (es la memoria que esta interna a una partición, pero no esta siendo usada).
Una solución al problema de la fragmentación externa es la compactación. Su objetivo es el de barajar los contenidos de la memoria para ubicar toda la memoria libre junta y formar un gran agujero. En la figura 8.10 se ve un ejemplo.

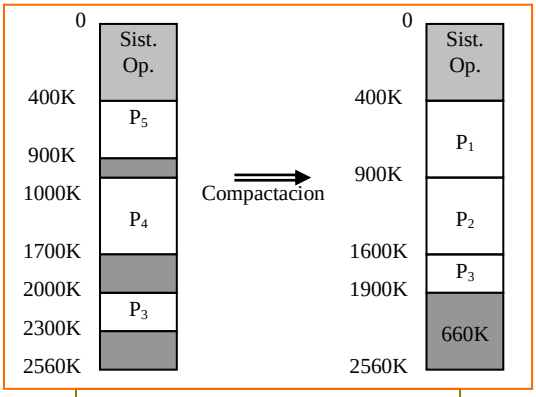


Figura 8.10 Compactación.

La compactación no siempre es posible. Notemos que en el ejemplo anterior los procesos P_4 y P_3 fueron movidos. Para que estos procesos sean capaces de ejecutar en sus nuevas direcciones, todas las direcciones internas deben ser cambiadas. Si la reubicación es estática y es realizada en tiempo de compilación o de carga, la compactación no se puede hacer. La compactación sólo es posible si la reubicación es dinámica, y es realizada en tiempo de ejecución. Si las direcciones son reubicadas dinámicamente, la reubicación requiere solo mover el programa y los datos, y luego cambiar el registro base para reflejar la nueva dirección base.
Swapping puede ser combinado con la compactación. Un proceso puede ser sacado de la memoria principal y ser llevado a un almacenamiento de seguridad y traído más tarde nuevamente. Cuando el proceso es sacado, su memoria es liberada, y quizá utilizada por algún otro proceso. Cuando el proceso esta listo para ser traído, pueden aparecer varios problemas. Si se usa reubicación estática, el proceso debe ser ubicado en la memoria en el mismo lugar que fue anteriormente sacado. Esta restricción puede provocar que otros procesos deban ser sacados para liberar dicha memoria. Si la reubicación es dinámica (con registros base y limite), entonces un proceso puede ser ubicado en un lugar diferente al anterior. En este caso, se debe buscar un bloque libre, compactando si es necesario, y ubicar el proceso en el bloque.

Paginación

Otra solución al problema de la fragmentación externa es la de permitir que el espacio de direcciones lógicas de un proceso sea no-contiguo. Una forma de implementar ésta solución es a través del uso del esquema de páginas.

Método básico: La memoria física es dividida en bloques de tamaño fijo llamados frames. La memoria lógica es también dividida en bloques del mismo tamaño llamados páginas. Cuando un proceso está para ser ejecutado, sus páginas son cargadas en cualquier frame de memoria disponible. El almacenamiento de seguridad es también dividido en bloques de tamaño fijo cuyos tamaños son iguales a los frames de la memoria.
El soporte de hardware para el paginado se ve en la figura 8.12. Cada dirección que la CPU genera se divide en dos partes: el número de página (p), y el offset de la página (d). El número de página es usado como

índice de la tabla de páginas. Esta tabla contiene la dirección base de cada página en la memoria física. Esta dirección base es combinada con el offset de la página para formar la dirección de memoria física que es enviada a la unidad de memoria. Un modelo de paginado de la memoria se ve en la figura 8.13.

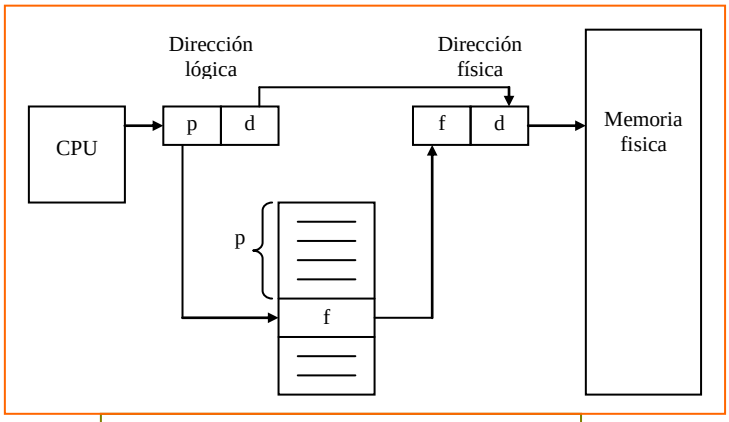


Figura 8.12 Hardware de paginado.

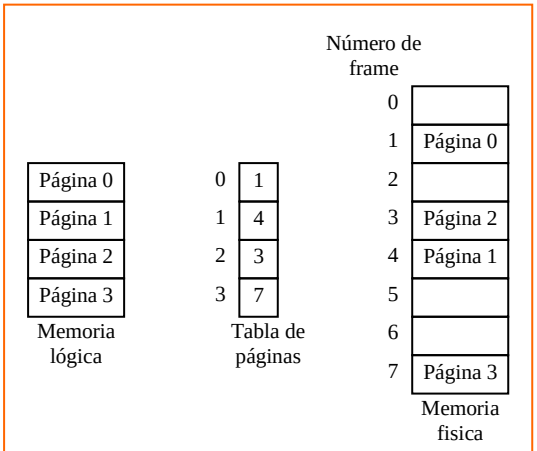


Figura 8.13 Modelo de paginado de la memoria lógica y física.

El tamaño de página (como el tamaño del frame) es definido por el hardware. El tamaño de una página es típicamente una potencia de 2 variando entre 512 bytes y 16 megabytes por página, dependiendo de la arquitectura de la computadora. La elección de porque es una potencia de 2 se debe a que la traslación desde una dirección lógica a un número de página y su offset sea muy fácil. Si el tamaño del espacio de la dirección lógica es 2^m, y el tamaño de la página es de 2^n unidades de direccionamiento (bytes o palabras), entonces los m-n bytes de orden alto de la dirección lógica designan el número de página, y los n bits de orden bajo designan el offset de la página.

Como un simple ejemplo, supongamos que tenemos páginas de 4 bytes, y una memoria física de 32 bytes (8 páginas) (Figura 8.14). La dirección lógica 0 esta en la página 0, offset 0. Analizando la tabla de páginas encontramos que la página 0 esta en el frame 5. Así, la dirección lógica 0 se mapea con la dirección física 20 (=(5*4)+0). La dirección lógica 3 (página 0, offset 3) se mapea con la dirección física 23 (=(5*4)+3). La dirección lógica 4, esta en la página 1, offset 0; de acuerdo a la tabla de páginas, la página 1 es mapeada al frame 6. Así, la dirección lógica 4 se mapea con la dirección física 24 (=(6*4)+0).

Note que el paginado es una forma de reubicación dinámica. Cada dirección lógica esta limitada por el hardware del paginado a alguna dirección física

Cuando usamos un esquema paginado, no se tiene fragmentación externa. Sin embargo, tenemos algo de fragmentación interna. Note que los frames son asignados como unidades, por lo que en caso de que el requerimiento de memoria de un proceso no es múltiplo del tamaño de página, entonces la ultima página quedara parcialmente llena. Por ejemplo, si las páginas son de 2048 bytes, un proceso de 72766 bytes requerirá 35 páginas y 1086 bytes, por lo que necesitara 36 frames resultando en una fragmentación interna de 2048 – 1086 = 962 bytes. En el peor caso, un proceso necesitara n páginas más 1 bytes, por lo que se le asignaran n + 1 frames, resultando en una fragmentación interna de casi un frame entero. Las páginas de hoy en día son de 2 a 4 K.

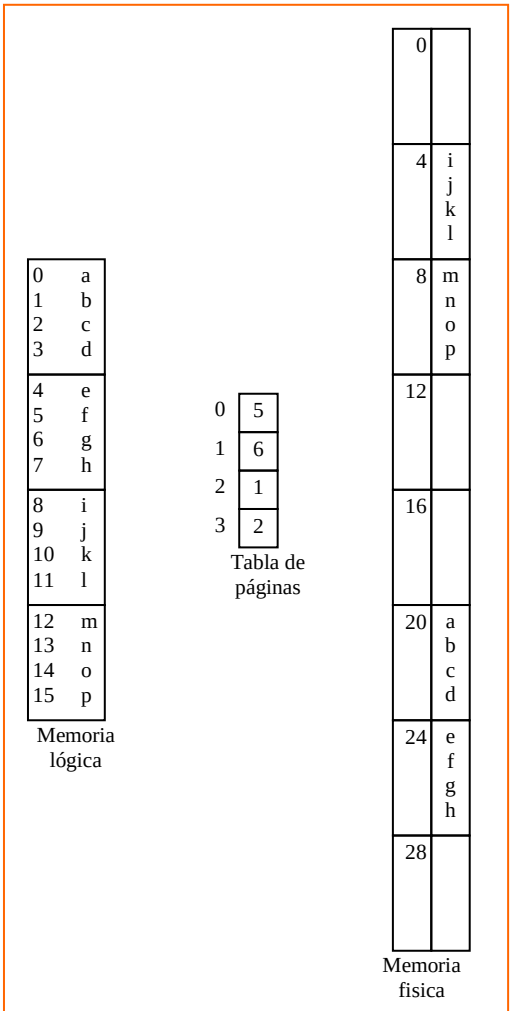


Figura 8.14 Ejemplo de paginado.

Cuando un proceso llega al sistema para ser ejecutado se examina su tamaño para ver cuantas páginas necesita. Cada página del proceso necesita un frame. Así, si el proceso requiere n páginas, debe haber al menos n frames disponibles en la memoria. En caso de que los haya, se le asignan al proceso. La primer página del proceso es cargada en un frame asignado, y el número de frame es puesto en la tabla de páginas para éste proceso. La siguiente página es cargada en otro frame, y su número de frame es puesto en la tabla, y así con todas las páginas (Figura 8.15).

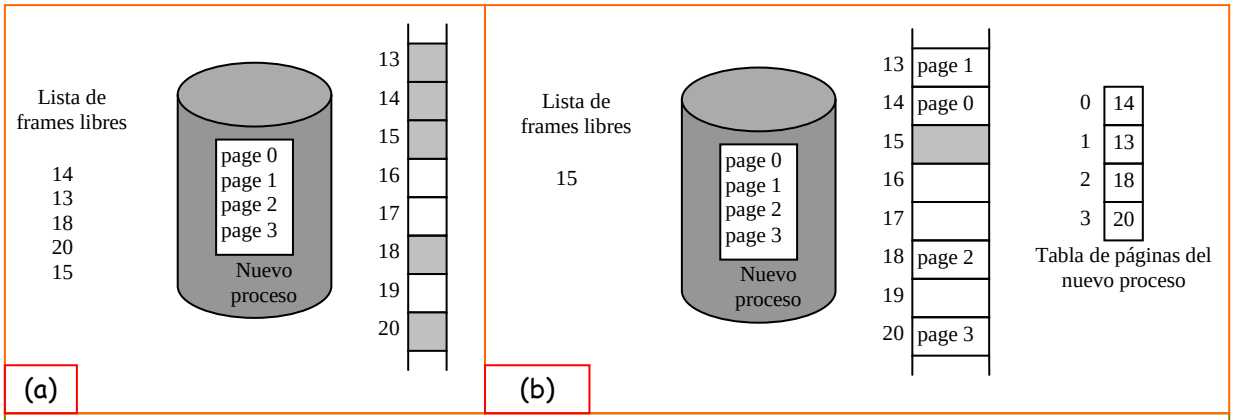


Figura 8.15 Frames libres. (a) Antes de la asignación. (b) Después de la asignación.

Ya que el sistema operativo es quien administra la memoria física, éste debe saber los detalles de asignación de la memoria física: cuales frames están asignados, cuales disponibles, cuantos frames existen, etc. Esta información es mantenida por una estructura de datos llamada tabla de frame. Esta tabla tiene una entrada para cada frame físico, indicando si esta disponible o no, y en caso de que este asignado, a que página de que proceso o procesos. Además, el sistema operativo debe tener información de cuales procesos operan en el espacio del usuario, y todas las direcciones lógicas deben ser transformadas en direcciones físicas. Si un proceso hace una llamada al sistema (por ejemplo, para hacer I/O) y provee una dirección como parámetro (por ejemplo, un buffer), ésta dirección también debe ser mapeada para producir la dirección física correcta. El sistema operativo mantiene una copia de la tabla de páginas de cada proceso, así como mantiene una copia del contador de instrucción y el contenido de los registros. Esta copia es usada por el dispatcher de CPU para definir el hardware de la tabla de página cuando se le asigna la CPU a un proceso. Ante esto, la paginación incrementa el tiempo de context-switch.

Estructura de la tabla de páginas: Cada sistema operativo tiene su propio método para almacenar la tabla de páginas. La mayoría asigna una tabla de páginas para cada proceso. Un puntero a la tabla de páginas es almacenado con otros valores de registro (tal como el contador de instrucción) en el PCB del proceso. Cuando el dispatcher comienza un nuevo proceso, debe recargar los registros del usuario y definir los valores de la tabla de páginas.

- *Soporte del hardware:* la implementación del hardware de la tabla de páginas puede ser hecho de una manera diferente de formas. En el caso más simple, la tabla de páginas es implementada como un conjunto de registros. Estos registros deben ser construidos con una alta velocidad lógica para hacer eficiente la traducción de las direcciones de paginado. Cada acceso a la memoria debe pasar a través del mapeo del paginado, por lo que el tema de la eficiencia es muy importante. El dispatcher de la CPU recarga estos registros, así como lo hace con la carga de los demás registros. Las instrucciones para cargar o modificar los registros de la tabla de páginas son obviamente privilegiadas, por lo que solo el sistema operativo puede llevarlas a cabo.
  La utilización de los registros para mantener la tabla de páginas es útil solo si dicha tabla es pequeña (256 entradas). Las actuales computadoras, sin embargo, permiten que la tabla de páginas sea muy grande (1 millón de entradas) por lo que el uso de registros rápidos para implementar la tabla de páginas en estas maquinas no es útil y dicha tabla se mantiene en la memoria principal, y un registro base de la tabla (*Page-Table Base Register: PTBR*) apunta a la tabla. Cambiar la tabla de páginas requiere solo cambiar un registro, reduciendo en gran parte el tiempo de context switch.
  El problema con este sistema es el tiempo que lleva acceder a un lugar de la memoria del usuario. Si queremos acceder a la posición i, primero debemos llegar a la tabla de páginas por medio del PTBR corrido por el número de página para i. Esta tarea requiere un acceso a memoria. Esto nos provee el número de frame, el cual combinado con el offset produce la actual dirección, pudiendo entonces acceder a la ubicación deseada de la memoria. Así, con este esquema se necesitan dos accesos a memoria para acceder al byte (una para la entrada a la tabla, la otra para acceder al byte).
  La solución estándar es la de usar una pequeña, pero muy rápida, cache, llamada *registros asociativos* o también *TLBs* (*translation look-aside buffer*). Éstos registros son una memoria de alta velocidad. Cada registro tiene dos campos: la clave y el valor. Por lo tanto, ante la existencia de un ítem, éste puede ser comparado con las claves de todos los registros asociativos simultáneamente (es decir, con todos de un solo paso). En caso de que el ítem se encuentre, se devuelve el correspondiente campo de valor. La búsqueda es muy rápida pero el hardware es muy caro. El número de entradas en un TLB varía entre 8 y 2048.
  Los registros asociativos son usados con la tabla de páginas de la siguiente forma. Los registros asociativos contienen solo algunas entradas de la tabla de páginas. Cuando la CPU genera una dirección lógica, el número de página se presenta ante la cache que contiene algunos números de páginas y sus correspondientes números de frames. Si el número de página esta en los registros asociativos, el número de frame es inmediatamente devuelto y puede ser usado para acceder a la memoria. En caso de que el número de página no este en la cache, se debe hacer una referencia al

resto de la tabla de páginas que esta en memoria. Al obtenerse el número de frame, se puede acceder a la memoria (Figura 8.16). Además, se agrega dicho número de página y número de frame a la tabla de páginas de la cache para así es encontrada en la siguiente referencia. Si el TLB esta completamente lleno, el sistema operativo debe seleccionar una entrada para reemplazarla. Desgraciadamente, cada vez que se selecciona una tabla de páginas (por ejemplo, cada context switch), el TLB debe ser vaciado (borrado) para asegurar que el siguiente proceso a ejecutar no use una traducción errónea.

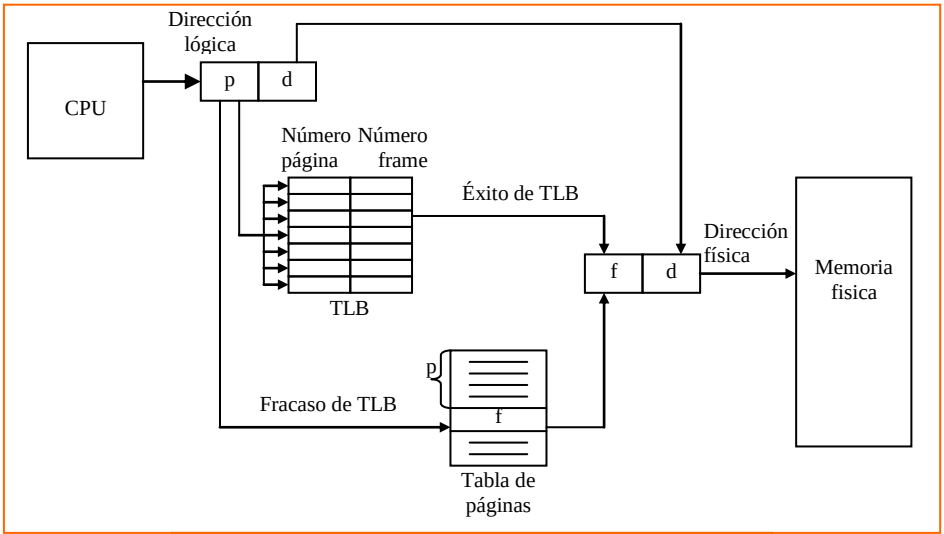


Figura 8.16 Hardware del paginado con TLB.

El porcentaje de veces que la página buscada es encontrada en la memoria cache se denomina *hit ratio* (proporción de éxito). Un hit ratio del 80% significa que encontramos el número de página deseado en la cache el 80% de las veces. Si el acceso a la cache toma 20 nanosegundos, y 100 nanosegundos toma el acceso a memoria, entonces un acceso a memoria mapeado toma 120 nanosegundos en caso de que el número de la página este en la cache. En caso de que fallemos en la búsqueda del número de página en la cache (20 nanosegundos), debemos realizar un primer acceso a la memoria para buscar el número de frame en la tabla de página (100 nanosegundos), y luego el acceso al byte deseado en memoria (100 nanosegundos), para un total de 220 nanosegundos. Para encontrar el tiempo de acceso efectivo hacemos:

Tiempo acceso efectivo = 0.8 * 120 + 0.2 * 220 = 140 nanosegundos.

- *Protección:* la protección de la memoria en un entorno paginado es llevada a cabo por bits que son asociados con cada frame. Normalmente estos bits son mantenidos en la tabla de páginas. Un bit puede definir si una página es de lectura-escritura o de solo lectura. Cada referencia a memoria atraviesa la tabla de páginas para encontrar el número de frame correcto. En el mismo momento que la dirección física esta siendo calculada, se chequean los bits de protección para verificar que el tipo de acceso es correcto. Un intento de escribir en una página de solo lectura causara un trap de hardware al sistema operativo (violación de la protección de memoria).
  Generalmente aparece un bit más en cada entrada de la tabla de páginas: un bit valido-invalido. Cuando se setea este bit en valido, este valor indica que la página asociada esta en el espacio de dirección lógica del proceso, por lo que dicha página es legal. Si el bit esta seteado a invalido, el valor indica que la página no esta en el espacio de dirección lógica del proceso. Las direcciones ilegales son atrapadas usando el bit de valido-invalido. El sistema operativo setea este bit para cada página para aprobar o no el acceso a la página. Un intento de leer una página que es invalida provocara un trap al sistema operativo (referencia de página invalida).
  Supongamos un ejemplo donde el sistema tiene un espacio de direcciones de 14 bits (0 a 16383) y se tiene un programa que usa solo las direcciones desde 0 a 10468 (Figura 8.17). Con un tamaño de

página de 2K, se le asignan al proceso las páginas 0, 1, 2, 3, 4 y 5. Note que el programa solo se extiende hasta la dirección 10468, cualquier referencia debajo de ésta debería ser tomada como ilegal. Sin embargo, toda referencia a la página 5 es valida, por lo que los accesos hasta la dirección 12287 son validos. Solo las direcciones desde 12288 hasta 16383 son invalidas.

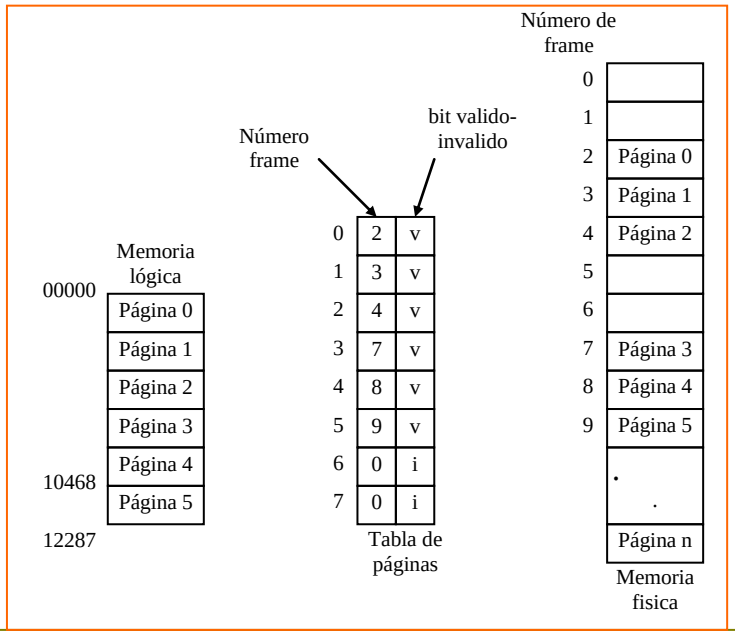


Figura 8.17 Bit válido (v) o inválido (i) en una tabla de página.

Este problema es el resultado de un tamaño de página de 2K y refleja la fragmentación interna del paginado. Raramente un proceso usa todo su rango de direcciones. De hecho, muchos procesos solo usan una pequeña fracción del espacio de direcciones disponible para ellos. Ante esto, provoca mucho gasto crear una tabla de páginas con entradas para cada página en el rango de direcciones. La mayor parte de la tabla estaría sin usar, pero tomaría lugar en la memoria. Algunos sistemas proveen hardware en la forma de un registro que indica el tamaño de la tabla de página (PTLR: page-table length register). Este valor es chequeado contra cada dirección lógica para verificar que la dirección este en el rango valido para el proceso. Un fallo de este test causara un trap al sistema operativo.

Paginado multinivel: la mayoría de los sistemas de computadoras actuales soportan un espacio de direcciones lógicas muy grande (desde 2^{32} a 2^{64}). En tales sistemas la tabla de páginas se convierte muy grande. Por ejemplo, consideremos un sistema con un espacio de direcciones lógicas de 32 bits. Si la página es de tamaño de 4K (2^{12}), entonces la tabla de páginas consistirá de más de 1 millón de entradas ($2^{32}/2^{12} = 2^{20} = 1.048.576$). Ya que cada entrada consiste de 4 bytes, cada proceso puede necesitar hasta 4 megabytes de espacio de dirección física sólo para la tabla de páginas (1.048.576 * 4 bytes = 4.194.304 bytes = 4096 Kb = 4 Mb). Una solución es dividir la tabla de páginas en pequeñas piezas.
Una forma es usar el esquema paginado de dos niveles, en el cual la tabla de páginas por si mismo también es paginada (Figura 8.18).

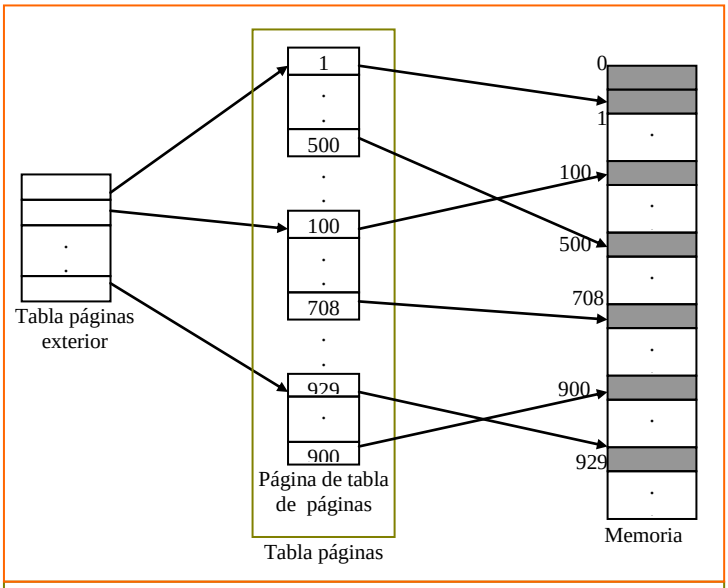
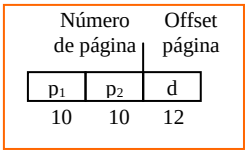


Figura 8.18 Esquema de tabla de página de dos niveles.

Para ilustrarlo, retornemos a nuestro ejemplo de 32 bits, con un tamaño de página de 4K. La dirección lógica se divide en un número de páginas de 20 bits, y el offset de 12 bits. Ya que paginamos la tabla de páginas, el número de páginas se divide además en un número de página de 10 bits, y un offset de página de 10 bits. Así, la dirección lógica se vería como:



donde p_1 es un índice de la tabla de páginas exterior, y p_2 es el desplazamiento en la página de la tabla de páginas exterior.
El esquema de traslación de direcciones se ve en la figura 8.19.

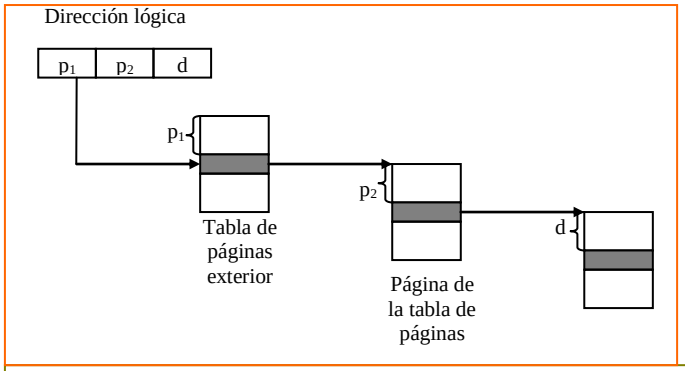
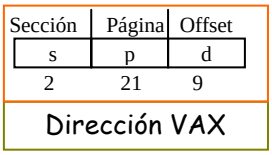


Figura 8.19 Traducción de direcciones para una arquitectura paginada de 32 bits de dos niveles.

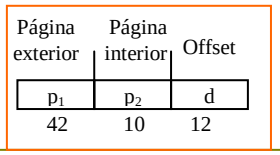
La arquitectura VAX soporta el paginado de dos niveles. VAX es una maquina de 32 bits con tamaño de página de 512 bytes. El espacio de dirección lógica de un proceso es dividido en cuatro secciones iguales cada una de 2^{30} bytes. Cada sección representa a una parte diferente del espacio de direcciones lógicas de un proceso. Los primeros 2 bits de orden más alto de la dirección lógica designan la sección apropiada. Los restantes 21 bits representan el número de página lógica de ésta sección, y los últimos 9 bits representan el offset en la página. Particionando la tabla de páginas de esta manera, el sistema operativo puede dejar

particiones que no se usan hasta que un proceso las necesite. Una dirección en la arquitectura VAX es como sigue:



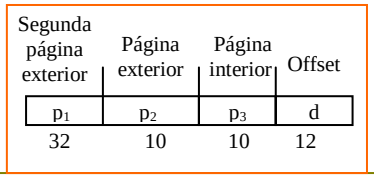
Dirección VAX

donde s designa el número de sección, p es el índice en la tabla de páginas, y d el desplazamiento en la página.
Para un sistema con un espacio de direcciones lógicas de 64 bits, un esquema de paginado de dos niveles no es apropiado. Para ilustrar el porque, supongamos que el tamaño de página en tal sistema es de 4K (2^{12}). En este caso, la tabla de páginas consistirá de hasta 2^{52} entradas. Si utilizamos un paginado de dos niveles, entonces las tablas de páginas internas podrían ser convenientemente de 1 página de largo, o contener 2^{10} entradas de 4 bytes. La dirección se vería como sigue:



Dirección no conveniente

La tabla de páginas exterior consistirá de 2^{42} entradas, o de 2^{44} bytes (cada entrada es de 4 bytes). El método obvio para evitar que tal tabla de páginas sea tan larga es la de dividir la tabla de página exterior en pequeñas piezas. Esta división se puede llevar a cabo de diferentes maneras. Podemos paginar la tabla de páginas exterior dando un esquema de paginado de tres niveles. Supongamos que la tabla de páginas exterior es hecha en páginas de tamaño estándar (2^{10} entradas, o 2^{12} bytes); un espacio de direcciones de 64 bits es aun desalentador, ya que la tabla de páginas exterior es aun de 2^{34} bytes de grande.



Dirección tampoco conveniente

El siguiente paso seria paginar nuevamente para llegar a un esquema de paginado de 4 niveles, donde la segunda página exterior seria también paginada.
Veamos como afecta el paginado al rendimiento del sistema. Ya que cada nivel es almacenado en una tabla separada en memoria, convertir una dirección lógica en una física puede tomar cuatro accesos a memoria. Ante esto, hemos quintuplicado el tiempo necesitado para acceder a la memoria. Con una cache de una taza de éxito del 98%, tenemos:

Tiempo de acceso efectivo = 0.98 * 120 + 0.02 * 520 = 128 nanosegundos.

Así, aun con todos los niveles extras de la tabla, solo tenemos un 28% de retraso en el tiempo de acceso a memoria.

Tabla de página invertida: Usualmente cada proceso tiene su propia tabla de páginas. La tabla de páginas tiene una entrada para cada página que el proceso esta usando (o mejor dicho, un lugar para cada dirección virtual, indiferente de si es valida o no). Esta representación de la tabla es natural, ya que los procesos se refieren a las páginas a través de direcciones virtuales de páginas. El sistema operativo debe traducir esta referencia a una dirección de memoria física. Ya que la tabla de páginas esta ordenada por direcciones virtuales, el sistema operativo es capaz de calcular donde esta la dirección física asociada con cada

dirección virtual y usar dicho valor directamente. Uno de los inconvenientes de este esquema es que cada tabla de página puede consistir de millones de entradas. Estas tablas pueden consumir grandes cantidades de memoria física, el cual solo se requiere para llevar una pista de cómo están siendo usada la otra parte de la memoria.

Para resolver este problema podemos usar la tabla de páginas invertida. En tales tablas se tiene una entrada para cada página real de memoria (frame). Cada entrada consiste de la dirección virtual de la página almacenada en ese lugar de memoria real, con información sobre el proceso dueño de la página. Así, existe solo una tabla de páginas en el sistema, y ésta solo tiene una entrada para cada página de la memoria física (Figura 8.20).

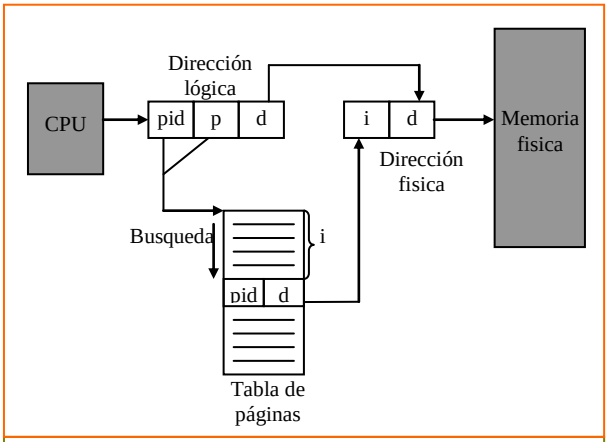


Figura 8.20 Tabla de página invertida.

Para ilustrar este esquema, se describirá como implementa el esquema de página invertida en IBM RT. Cada dirección virtual del sistema consiste de una tripla:

<id-proceso, número-página, offset>.

Cada entrada de la tabla de página invertida es un par:

<id-proceso, número-página>.

Cuando ocurre una referencia a memoria, parte de la dirección virtual consistentes de <id-proceso, número-página> se presenta en el subsistema de la memoria. La tabla de páginas invertida es entonces recorrida para encontrar una coincidencia. Si se encuentra dicha coincidencia (sea la entrada número i), entonces la dirección física es <i, offset>. En caso de no encontrar una coincidencia, entonces se ha intentado hacer una acceso a una dirección ilegal. Aunque este esquema decrece la cantidad de memoria necesitada para almacenar cada tabla de página, incrementa la cantidad de tiempo necesitado para recorrer la tabla cuando ocurre una referencia a memoria. Ya que la tabla esta ordenada por una dirección física, pero la búsqueda ocurre sobre direcciones virtuales, se necesita rastrear toda la tabla para encontrar una coincidencia. Esta búsqueda tomara gran cantidad de tiempo. Para aliviar este problema, se utiliza una tabla de hash para limitar la búsqueda. Por supuesto que cada acceso a la tabla de hash agrega una referencia a memoria al procedimiento, por lo que una referencia a memoria virtual requiere por lo menos dos lecturas a la memoria real: una para la entrada a la tabla de hash, y la otra para la tabla de páginas. Para mejorar el rendimiento, se utiliza registros de memoria asociativos que mantienen recientes entradas ubicadas. Estos registros son rastreados primero, antes de que sea consultada la tabla de hash.

Páginas compartidas: otra ventaja del paginado es el de compartir código en común. Esta consideración es importante en los entornos de tiempo compartido. Consideremos un sistema que soporta 40 usuarios, cada uno ejecutando un editor de texto. Si el editor de texto consiste de 150K de código y 50K de datos, necesitaríamos 8000K para soportar los 40 usuarios. Sin embargo, si el código es reentrante (solo lectura) entonces puede ser compartido. En la figura 8.21 se ve dicho esquema, donde hay un editor de tres páginas (cada una de 50K; el tamaño fue usado de 50K para simplificar la figura), donde cada página esta siendo compartida por tres procesos. Cada proceso tiene su propia página de datos. El código reentrante (también

llamado código puro) es aquel código que no puede sufrir modificaciones. Si el código es reentrante, entonces nunca sufre cambios durante la ejecución. Así, dos o más procesos pueden ejecutar el mismo código a la vez. Cada proceso tiene su propia copia de registros y datos para mantener información del proceso en ejecución. Obviamente, la parte de datos de cada proceso será diferente y variara durante la ejecución.

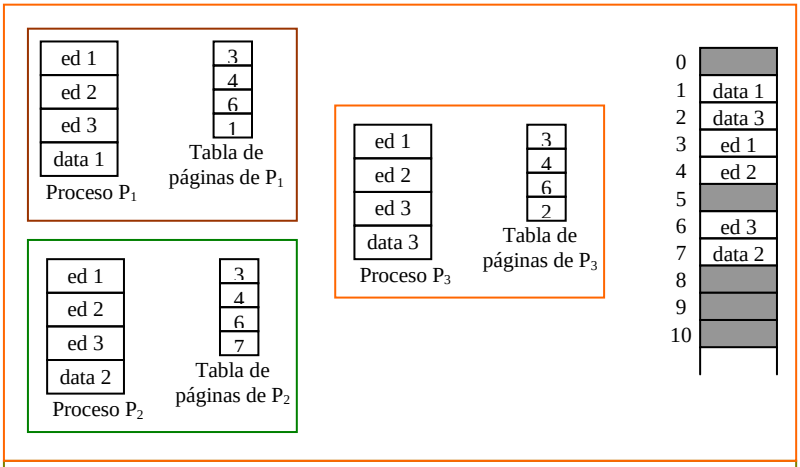


Figura 8.21 Compartiendo código en un ambiente paginado.

Solo se necesita una copia del editor en memoria. Cada tabla de la página del usuario se mapea con la misma copia física del editor, pero las páginas de los datos son mapeadas en frames diferentes. Así, para soportar 40 usuarios, solo necesitamos una copia del editor (150K), más 40 copias del espacio de datos de cada proceso. El espacio total requerido es ahora de 2150.

Otros programas muy usados también pueden ser compartidos: compiladores, sistemas windows, sistemas de base de datos, etc. Para que pueda ser compartido, el código debe ser reentrante. Esta forma de compartir entre los procesos en un sistema es similar a la forma en como comparten los threads el espacio de direcciones de una tarea.

Los sistemas que utilizan paginación invertida tienen dificultad para implementar memoria compartida. La memoria compartida es usualmente implementada como dos direcciones virtuales que son mapeadas en una misma dirección física. Sin embargo, éste método no puede ser usado cuando hay solo una entrada de página virtual para cada página física, de manera que una página física no puede tener dos (o más) direcciones virtuales compartidas.

Segmentación

Un aspecto importante de la administración de memoria que era inevitable con paginación es la separación de la vista que el usuario tenia de la memoria y la actual memoria física. La vista del usuario de la memoria no es la misma de la actual memoria física. La vista del usuario es mapeada sobre la memoria física. El mapeo permite diferenciar entre la memoria lógica y la memoria física.

Método básico: El usuario o el programador imaginan a la memoria como un arreglo lineal de bytes. Más bien, el usuario prefiere ver a la memoria como una colección de segmentos de tamaño variables, sin la necesidad de que estos segmentos estén ordenados (Figura 8.22).

Supongamos que usted esta escribiendo un programa. Usted lo ve como un programa principal con un conjunto de subrutinas, procedimientos, funciones o módulos. Puede haber también varias estructuras de datos: arreglos, tablas, pilas, variables, etc. Cada uno de estos módulos o elementos de datos es referido por un nombre. Usted vera el elemento "aux", la función "Sqrt", pero no tendrá en cuenta en ningún momento en que dirección de memoria se encuentra. Usted no sabe si la variable fue almacenada antes o después de la función. Cada uno de estos segmentos es de tamaño variable. Los elementos en un segmento

están identificados por su offset desde el comienzo del segmento.: la primer sentencia del programa, la décima entrada de la tabla de símbolos, la quinta instrucción de la función Sqrt, etc.

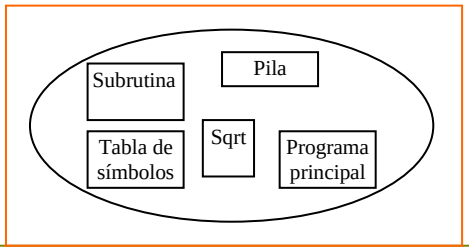


Figura 8.22 Vista del usuario de un programa.

La segmentación es un esquema de administración de memoria que soporta esta vista del usuario. Un espacio de dirección lógica es una colección de segmentos. Cada segmento tiene un nombre y tamaño. La dirección especifica tanto el nombre del segmento como el offset en el segmento. Ante esto, el usuario especifica cada dirección como: el nombre y el offset. En la paginación el usuario solo especificaba un número único, el cual era particionado por el hardware en un número de página y un offset, todo invisible para el programador.

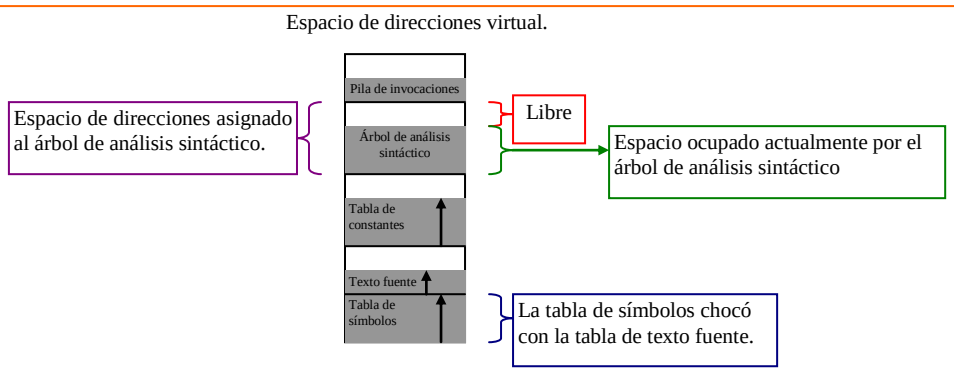
Por simplicidad de implementación, los segmentos son numerados y son referidos por dicho número, en lugar de por el nombre. Así, una dirección lógica consiste de:

<número-de-segmento, offset>.

Hasta ahora se vio a la memoria como un arreglo unidireccional ya que las direcciones virtuales van desde 0 hasta alguna dirección máxima, una dirección tras otra. Para muchos problemas, tener dos o más espacios de direcciones virtuales independientes puede ser mucho mejor que tener sólo uno. Por ejemplo, un compilador tiene muchas tablas que se construyen conforme procede la compilación, tales como:

1. El texto fuente.
2. La tabla de símbolos, que contiene los nombres y los atributos de las variables.
3. La tabla que contiene todas las constantes enteras y de punto flotante empleadas.
4. El árbol de análisis sintáctico, que contiene el árbol de análisis sintáctico del programa.
5. La pila empleada para llamadas a procedimientos dentro del compilador

El cargador tomara cada uno de estos segmentos y les asignara el número de segmento. Las primeras cuatro tablas crecen continuamente a medida que avanza la compilación. La última crece y se encoge de forma impredecible durante la compilación. En una memoria unidimensional, habría que asignar a éstas cinco tablas trozos contiguos del espacio de direcciones virtual, como se ve en la siguiente figura.



En un espacio de direcciones unidireccional con tablas crecientes, una tabla podría chocar contra otra.

Consideremos lo que ocurre si un programa tiene un número muy grande de variables. El trozo de espacio de direcciones asignado a la tabla de símbolos podría llenarse, pero podría haber espacio de sobra para

otras tablas. Desde luego, el compilador podría limitarse a exhibir un mensaje diciendo que la compilación no pudo terminar debido a un exceso de variables, pero no parece justo si existe espacio desocupado en las otras tablas.
Otra posibilidad sería sacarle espacio a las tablas que lo desperdician y dárselo a la tabla de símbolos. Es posible efectuar estos movimientos, pero es un trabajo muy costos.
Lo que realmente se necesita es una forma de liberar al programador de la tarea de administrar las tablas en expansión y contracción. Una solución directa sería proveer a la máquina con muchos espacios de direcciones completamente independientes, llamados **segmentos**. Cada segmento consiste de una secuencia lineal de direcciones, desde 0 hasta algún máximo. La longitud de cada segmento puede ser cualquiera desde 0 hasta el máximo permitido. Los diferentes segmentos pueden, y suelen, tener diferentes longitudes. Es más, la longitud de los segmentos puede cambiar durante la ejecución. La longitud de un segmento de pila puede aumentarse cada vez que algo se ingresa a la pila y disminuir cuando algo se saca.
Puesto que cada segmento constituye un espacio de direcciones aparte, los distintos segmentos pueden crecer y encogerse de forma independiente, sin afectarse entre sí. Si una pila de cierto segmento necesita más espacio de direcciones para crecer, se le puede conceder, porque no hay más en su espacio de direcciones con lo que pueda chocar. Desde luego, un segmento se podría llenar, pero los segmentos suelen ser muy grandes, por lo que esto es poco común.
Se subraya que un segmento es una unidad lógica, de la cual el programador está consciente y que utiliza como entidad lógica. Un segmento podría contener un procedimiento, un arreglo, una pila o una colección de variables, pero por lo general no contiene una mezcla de cosas.
Las memorias segmentadas tienen otras ventajas además de simplificar el manejo de estructuras de datos que están creciendo y encogiéndose. Si cada procedimiento ocupa un segmento aparte, con la dirección 0 como la dirección inicial, se simplifica considerablemente la vinculación de procedimientos compilados por separado. Una vez que se han compilado y vinculado todos los procedimientos que constituyen un programa, una llamada al procedimiento del segmento n usará la dirección de dos partes (n, 0) para direccionar la palabra 0 (el punto de entrada).
Si el procedimiento que está en el segmento n se modifica y recompila subsecuentemente, no hay necesidad de alterar ningún otro procedimiento (porque no se modifico ninguna dirección de inicio), incluso, si la nueva versión es más grande que la anterior. Con una memoria unidimensional, los procedimientos se empaquetan uno junto a otro, sin espacio de direcciones entre ellos. En consecuencia, la modificación del tamaño de un procedimiento puede afectar la dirección de inicio de otros procedimientos que no tienen relación con él. Esto, a su vez, requiere la modificación de todos los procedimientos que invocan a cualquiera de los procedimientos que se movieron, incorporando sus nuevas direcciones de inicio. Si un programa contiene cientos de procedimientos, este proceso puede ser costoso.
La segmentación también facilita compartir procedimientos o datos entre varios procesos. Un ejemplo común es la **biblioteca compartida**. Las estaciones de trabajo modernas que ejecutan sistemas de ventanas avanzados suelen tener bibliotecas de gráficos extremadamente grandes compiladas en casi todos sus programas. En un sistema segmentado, la biblioteca de gráficos se puede colocar en un segmento compartido por múltiples procesos, eliminando la necesidad de tenerlas en el espacio de direcciones de cada proceso. Si bien también es posible tener bibliotecas compartidas en los sistemas con paginación puro, resulta mucho más complicado. De hecho, estos sistemas lo logran simulando segmentación.
Dado que cada segmento constituye una entidad lógica de la cual el programador esta consciente, como un procedimiento, un arreglo, una pila, los diferentes segmentos pueden tener diferentes tipos de protección. Un segmento de procedimientos se puede especificar como sólo de ejecución, prohibiendo los intentos de leer o escribir sobre él.
El contenido de una página es, en cierto sentido, accidental. El programador no está consciente siquiera de que esta ocurriendo la paginación. Aunque sería posible colocar unos cuantos bits en cada entrada en la tabla de páginas para especificar el acceso permitido, para aprovechar esta capacidad el programador debería saber donde están las fronteras de página dentro de su espacio de direcciones. Ésta es precisamente la clase de administración que se intenta eliminar al inventar la paginación. Como el usuario de una memoria segmentada tiene la ilusión de que todos los segmentos están en la memoria principal todo

el tiempo, es decir, puede direccionarlos como que así fuera, puede proteger cada segmento individualmente, sin tener que preocuparse por la administración que implica superponerlos.

Consideración	Paginación	Segmentación
¿El programador necesita estar consciente de que se está usando ésta técnica?	No	Sí
¿Cuántos espacios de direcciones hay?	1	Muchos
¿El espacio de direcciones total puede exceder el tamaño de la memoria física?	Sí	Sí
¿Se puede distinguir los procedimientos de los datos y protegerse de forma independiente?	No	Sí
¿Se puede manejar fácilmente tablas cuyo tamaño fluctúa?	No	Sí
¿Se facilita la compartición de procedimientos entre usuarios?	No	Sí
¿Por qué se invento esta técnica?	Para tener un espacio de direcciones grande sin tener que adquirir más memoria física.	Para poder dividir los programas y los datos en espacios de direcciones lógicamente independientes y facilitar la compartición y protección.

Hardware: Aunque el usuario no se pueda referir al objeto en el programa por medio de una dirección de dos dimensiones, la memoria física es aun una secuencia de bytes de una dimensión. Así, debemos definir una implementación para mapear la dirección del usuario de dos dimensiones en una dirección física de una sola dimensión. Este mapeo lo efectúa la tabla de segmentos. Cada entrada de ésta tabla tiene una base del segmento y un limite del segmento. La base del segmento contiene la dirección física de comienzo donde el segmento reside en memoria, mientras que el limite del segmento especifica el largo del segmento.

El uso de la tabla de segmentos se ve en la figura 8.23. Una dirección lógica consiste de dos partes: el número de segmento *s*, y un offset *d* en el segmento. El número de segmento es usado como un índice en la tabla de segmentos. El offset d de una dirección lógica debe estar entre 0 y el limite del segmento. Si no es así, se envía un trap al sistema operativo. Si el offset es legal, es sumado con el valor de base del segmento para producir la dirección en la memoria física del byte deseado. Ante esto, la tabla de segmentos es esencialmente un arreglo de pares de registros base-limite.

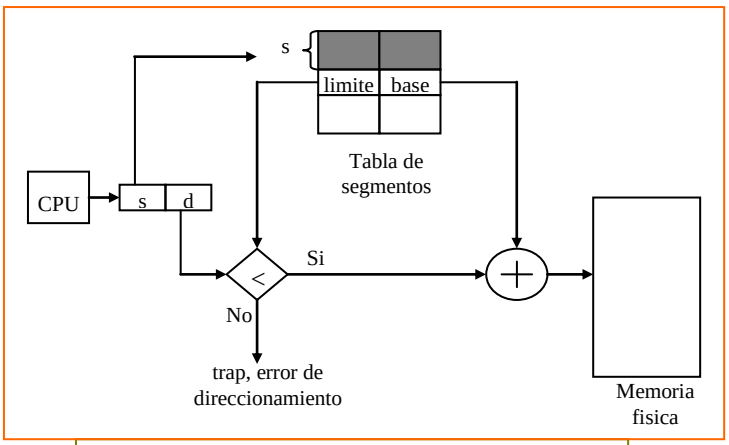


Figura 8.23 Hardware de segmentación.

Como ejemplo, consideremos la situación que se ve en la figura 8.24. Se tienen 5 segmentos numerados de 0 a 4. Los segmentos están almacenados en la memoria física como se muestra. La tabla de segmentos tiene una entrada separada para cada segmento, dando la dirección de comienzo del segmento en la memoria física (la base) y el largo del segmento (el límite). Por ejemplo, el segmento 2 es de 400 bytes de longitud, y comienza en el lugar 4300. Así, una referencia al byte 53 del segmento 2 es mapeado al lugar 4300 + 53 = 4353. Una referencia al segmento 3, byte 852, es mapeado al lugar 4052 (3200 + 852). Una referencia al byte 1222 del segmento 0 resultara en un trap al sistema operativo, ya que este segmento es de solo 1000 bytes de longitud.

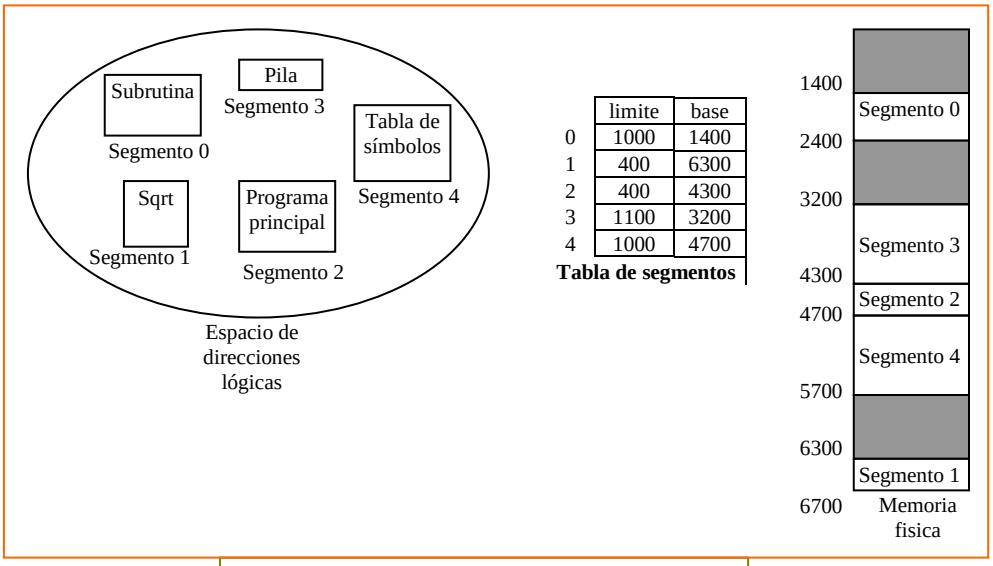


Figura 8.24 Ejemplo de segmentación.

Implementación de la tabla de segmentos: como la tabla de páginas, la tabla de segmentos puede ser puesta tanto en registros asociativos como en la memoria. Una tabla de segmentos mantenida en cache puede ser referenciada rápidamente; además, agregar la base al offset y comparar para que no se salga del rango se puede hacer simultáneamente para ahorrar tiempo.
En el caso que un programa consista de un gran número de segmentos, no es conveniente tener toda la tabla en cache, por lo que se debe mantener en memoria. Un registro de la base de la tabla de segmentos (*STBR: segment-table base register*) apunta a la tabla de segmentos. También, ya que el número de segmentos utilizado por un programa puede variar mucho, se utiliza un registro del largo de la tabla de segmentos (*STLR: segment-table length register*). Para una dirección lógica (s, d), primero se chequea que el número de segmento s es legal (es decir, s < STLR). Luego sumamos el número de segmento s con STBR (s + STBR), resultando en la dirección en memoria de la entrada de la tabla de segmentos. Esta entrada es leída de la memoria y procesada como antes: chequeamos el offset contra el largo del segmento y calculamos la dirección física del byte deseado como la suma de la base del segmento y el offset.
Como ocurre con páginas, este mapeo requiere dos accesos a memoria por dirección lógica. Una solución es usar un conjunto de registros asociativos para mantener las entradas de la tabla de segmentos más recientemente usadas.

Protección y compartición: una ventaja particular de la segmentación es la asociación de protección con los segmentos. Ya que los segmentos representan una porción definida semánticamente de un programa, es probable que todas las entradas en el segmento sean usadas de la misma forma. Así, tenemos algunos segmentos que son instrucciones, mientras que otros segmentos son datos. En una arquitectura moderna, las instrucciones no pueden ser modificadas, por lo que dichos segmentos pueden ser definidos de solo lectura o de solo ejecución. El hardware de mapeo de memoria chequeara los bits de protección asociados con cada entrada de la tabla de segmentos para prevenir accesos ilegales a memoria, tal como el intento de escribir en un segmento de solo lectura, o usar un segmento de solo ejecución como dato. Note que todos estos problemas son detectados por el hardware.
Otra ventaja de la segmentación involucra el compartir código o dato. Cada proceso tiene una tabla de segmentos asociada con él, el cual el dispatcher usa para definir el hardware de la tabla de segmentos cuando se le otorga la CPU a este proceso. Los segmentos son compartidos cuando entradas en las tablas de segmentos de dos procesos diferentes apuntan al mismo lugar físico (Figura 8.25).

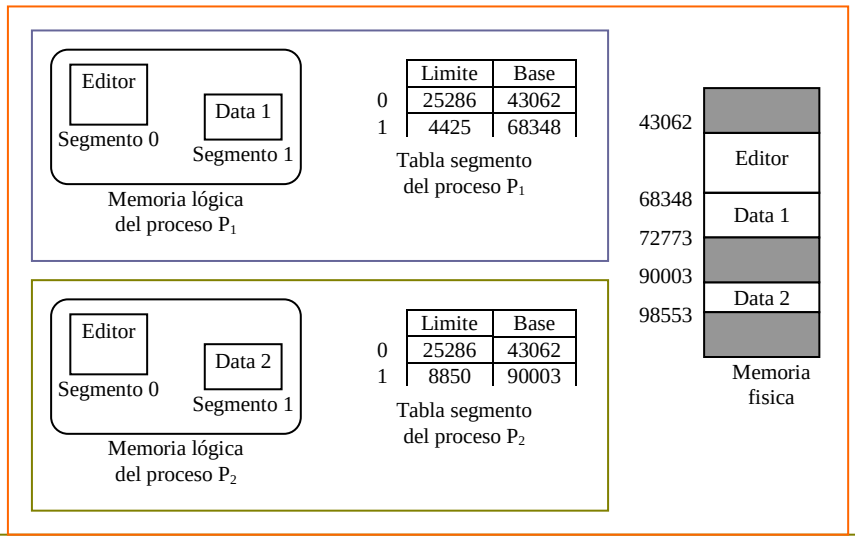


Figura 8.25 Compartición de segmentos en un sistema de memoria segmentada.

La compartición ocurre a nivel de segmento. Por ejemplo, consideremos el uso de un editor de texto en un sistema de tiempo compartido. Un editor completo puede ser bastante grande, compuesto por muchos segmentos. Estos segmentos pueden ser compartidos por muchos usuarios, limitando la memoria física necesitada para soportar las tareas del editor. En lugar de n copias del editor, necesitamos solo una copia. Para cada usuario, aun necesitamos separar los segmentos de variables locales. Obviamente estos segmentos no pueden ser compartidos.
También es posible compartir solo partes de programas. Por ejemplo, paquetes de subrutinas comunes pueden ser compartidas por muchos usuarios si son definidos como segmentos de solo lectura compartibles. Por ejemplo, dos programas FORTRAN pueden usar la misma función Sqrt, por lo que solo se necesitara una copia física de Sqrt.
Aunque esta compartición parece simple, hay varias consideraciones. Los segmentos de código típicamente contienen referencias a sí mismos. Por ejemplo, un salto condicional normalmente tiene una dirección de traslado. Esta dirección de traslado es un número de segmento y un offset. El número de segmento de la dirección de traslado será el número de segmento del segmento de código. Si tratamos de compartir este segmento, todos los procesos que lo compartan deben definir al segmento de código compartido con el mismo número de segmento. Por ejemplo, si queremos compartir la función Sqrt, y un proceso decide que dicha función esta en el segmento 4, mientras que otro proceso decide que esta en el segmento 17, ¿cómo hará la rutina Sqrt para referirse a ella misma?. Ya que hay una sola copia física de Sqrt, se debe referir a ella misma por un único nombre (número) de segmento para todos los usuarios. A medida que el número de usuarios que comparten segmentos aumenta, se hace más difícil encontrar un número de segmento adecuado.
Los segmentos de datos de solo lectura que no contienen punteros físicos pueden ser compartidos como números de segmentos diferentes, así como pueden también los segmentos de código que se refieren a ellos mismo no directamente sino solo indirectamente. Por ejemplo, los branches condicionales que especifican la dirección de salto como un offset desde el contador de programa actual.

Fragmentación: el scheduler de largo plazo debe encontrar y asignar memoria para todos los segmentos del programa del usuario. Esta situación es similar al paginado excepto que los segmentos son de un tamaño variable; las páginas son todas del mismo tamaño. Así, la asignación de memoria es un problema de asignación de almacenamiento dinámico, usualmente resuelto con los algoritmos best-fit o first-fit.
Ante esto, la segmentación puede causar fragmentación externa, cuando todos los bloques de memoria libre de la memoria son demasiados pequeños para poder alojar un segmento. En este caso, el proceso puede simplemente tener que esperar hasta que esté disponible más memoria, o se utilice la compactación para poder formar un agujero suficientemente grande.

Segmentación con paginado

Ya sea el paginado o la segmentación, ambos tienen sus ventajas y desventajas. De hecho, los dos microprocesadores más utilizados, el Motorola 68000 esta diseñado basado en un espacio de direcciones chato, mientras que la familia Intel 80X86 esta basada en la segmentación. Ambos tienen un sistema de administración de memoria con una mezcla de segmentación y paginación. Esta combinación se ve claramente en los siguientes dos ejemplos.

MULTICS: en el sistema MULTICS, una dirección lógica esta formada por 18 bit que describen el número de segmento, y 16 bits que describen el offset. Aunque este esquema crea un espacio de direcciones de 34 bits, el overhead debido a la tabla de segmentos es tolerable. Sin embargo, con segmentos de 64K palabras, el tamaño promedio de los segmentos podría ser grande y la fragmentación externa podría ser un problema. Aun si la fragmentación externa no fuera un problema, el tiempo de búsqueda para asignar un segmento, usando first-fit o best-fit, podría ser grande. Así, debemos gastar tiempo debido a la fragmentación externa, o gastar tiempo debido a los buscadores, o ambos.
La solución adoptada fue la de paginar los segmentos. La paginación elimina la fragmentación externa y hace trivial el problema de asignación: cualquier frame vacío puede ser usado. Cada página en MULTICS consiste de 1K palabra. Así, el offset del segmento (16 bits) es desarmado en un número de página (6 bits) y un offset de página (10 bits). El número de página es un índice en la tabla de página para dar el número de frame. Finalmente, el número de frame se combina con el offset de la página para formar una dirección física. El esquema de traducción se ve en la figura 8.26. Note que la diferencia entre ésta solución y la segmentación pura es que las entradas de la tabla de segmentos contienen no la dirección base del segmento, sino la dirección base de la tabla de página para este segmento.

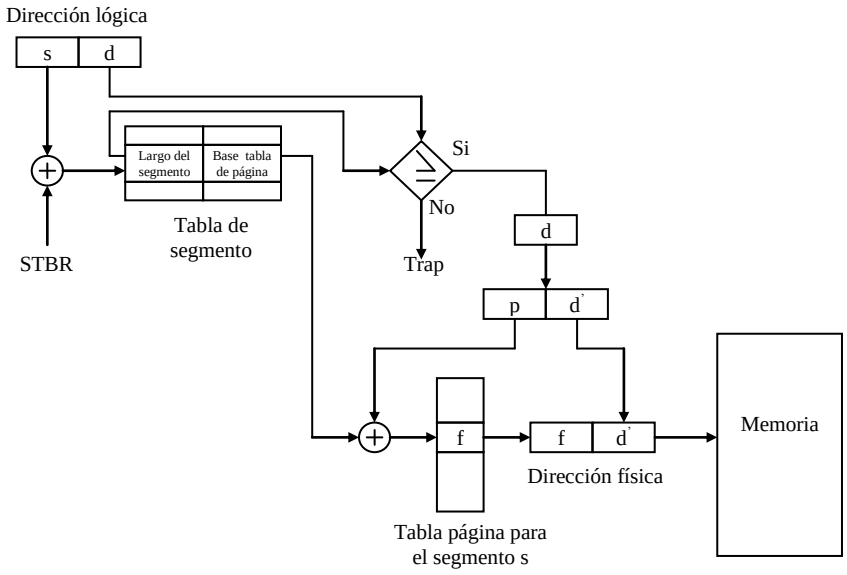
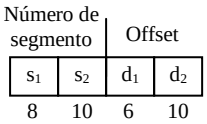


Figura 8.26 Segmentación paginada sobre el GE 645 (MULTICS).

Con esta solución se debe tener una tabla de página para cada segmento. Sin embargo, ya que cada segmento es limitado en el largo por su entrada de la tabla de página, la tabla de página no necesita estar llena. Esta requiere solo tantas entradas como necesite. Como con paginado, la última página de cada segmento generalmente no estará completamente llena.
En verdad, la segmentación paginada de MULTICS presentada es una simplificación de la real. Ya que el número de segmento es una cantidad de 18 bits, podemos tener más de 262144 segmentos (2^{18}), requiriendo un largo de la tabla de segmentos excesivamente grande. Para facilitar este problema, MULTICS página la tabla de segmentos. El número de segmento (18 bits) es dividido en un número de

página de 8 bits y un offset de página de 10 bits. Ahora, la tabla de segmento es representada por una tabla de página consistente de más de 2^8 entradas. Así, en general, una dirección lógica en MULTICS es:



donde s_1 es un índice a la tabla de página de la tabla de segmento y s_2 es el desplazamiento en la tabla de página de la tabla del segmento. Así, ya se ha encontrado la página conteniendo la tabla de segmentos deseada. Entonces, d_1 es el desplazamiento en la tabla de página del segmento deseado, y finalmente d_2 es el desplazamiento en la página conteniendo la palabra a ser accedida (ver Figura 8.27).

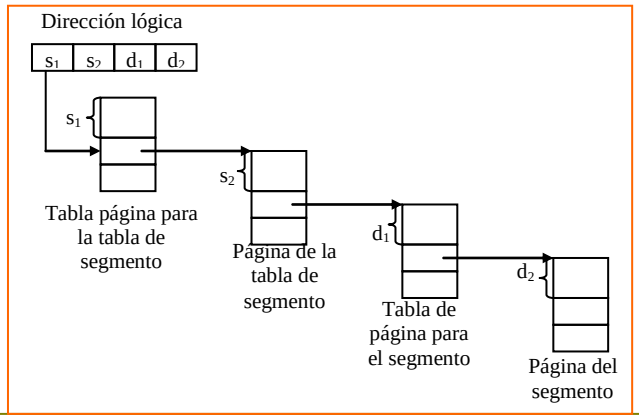


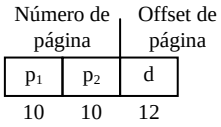
Figura 8.27 Traducción de dirección en MULTICS.

Para asegurar un rendimiento razonable, están disponibles 16 registros asociativos que contienen la dirección de las 16 referencias de páginas más recientes. Cada registro consiste de dos partes: una clave y un valor. La clave es un campo de 24 bits que es la concatenación de un número de segmento y un número de página. El valor es el número de frame.

Versión OS/2 32-Bit: La versión IBM de 32 bits es un sistema operativo corriendo en la arquitectura de Intel 386 (y siguientes). La 386 usa segmentación con paginado para la administración de memoria. El número máximo de segmentos por proceso es de 16K, y cada segmento puede ser tan grande como 4 gigabytes. El tamaño de página es de 4K bytes. Aquí se presentaran las ideas principales.
El espacio de direcciones lógicas de un proceso es dividido en dos particiones. La primer partición consiste de segmentos de más de 8K que son privados a este proceso. La segunda partición consiste de segmentos de más de 8K que son compartidos entre todos los procesos. La información sobre la primer partición se mantiene en la tabla descriptor local (LDT); la información sobre la segunda partición se mantiene en la tabla descriptor global (GDT). Cada entrada en la tabla LDT y en el GDT consiste de 8 bytes, con información detallada sobre un segmento en particular incluyendo la base y el largo del segmento.
La dirección lógica es un par (selector, offset), donde el selector es un número de 16 bits:

en el cual s designa el número de segmento, g indica si el número de segmento esta en GDT o en LDT, y p son de protección. El offset es un número de 32 bits especificando el lugar del byte (word) en el segmento en cuestión.
La maquina tiene 6 registros de segmento, permitiendo que los 6 segmentos sean direccionados en cualquier momento por un proceso. Este tiene seis registros microprogramados de 8 bytes para mantener el descriptor correspondiente ya sea de LDT o de GDT. Esta cache evita al 386 tener que leer el descriptor de la memoria para cada referencia a memoria.

La dirección física en el 386 es de 32 bits y esta formada como sigue. El registro seleccionado apunta a la entrada apropiada en LDT o GDT. La información de la base y limite sobre el segmento en cuestión e usada para generar una dirección lineal. Primero, el límite es usado para chequear la validez de la dirección. Si la dirección no es valida, se genera un fallo de memoria, resultando en un trap al sistema operativo. En caso de que sea valida, se agrega el valor del offset al valor de la base, resultando en una dirección lineal de 32 bits. Esta dirección es luego traducida a una dirección física.
Como se dijo anteriormente, cada segmento es paginado, y cada página es de 4K bytes. Una tabla de página puede entonces consistir de más de un millón de entradas. Ya que cada entrada consiste de 4 bytes, cada proceso puede necesitar más de 4 megabytes de espacio de memoria física solo para la tabla de página. Claramente, no deseamos mantener toda la tabla de páginas en la memoria. La solución adoptada es usar un esquema de paginado de dos niveles. La dirección lineal es dividida en un número d página consistente de 20 bits, y un offset de página consistente de 12 bits. Ya que paginamos la tabla de páginas, el número de página es además dividido en un puntero al directorio de página de 10 bits y un puntero a la tabla de página de 10 bits. La dirección lógica es entonces:



Ya que la eficiencia del uso de la memoria física puede ser mejorada, las tablas de páginas del Intel 386 pueden ser swapped a disco. En este caso, un bit de invalides es usado en la entrada del directorio de página para indicar si la tabla a la cual ésta entrada esta apuntando esta en disco o en memoria. Si la tabla esta en disco, el sistema operativo puede usar los otros 31 bits para especificar el lugar del disco donde esta la tabla; luego la tabla puede ser traída a memoria por demanda.

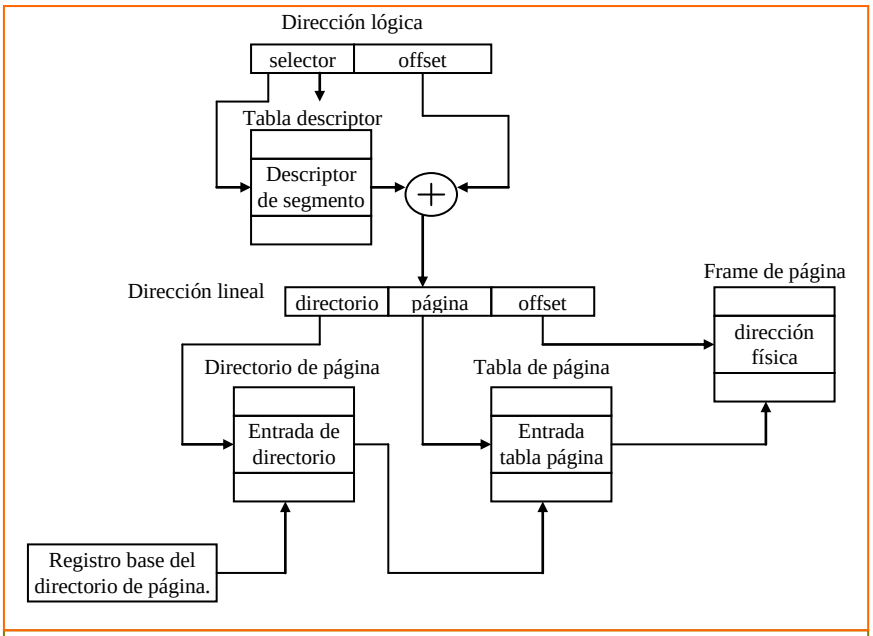


Figura 8.28 Traducción de dirección en Intel 8.28

9. Memoria Virtual

Para permitir que la CPU sea compartida por varios procesos, se deben mantener en memoria varios procesos, por lo que debemos compartir memoria. En este capítulo, veremos varios mecanismos para administrar la memoria.

La mayor parte de los sistemas de memoria virtual emplean la paginación. En cualquier computadora, existe un conjunto de direcciones de memoria que los programas pueden producir. Cuando un programa ejecuta una instrucción como:

MOVE REG, 1000

está copiando el contenido de la dirección 1000 en REG (o viceversa dependiendo de la computadora). Estas direcciones generadas por programas se denominan **direcciones virtuales** y forman el **espacio de direcciones virtuales**. En una computadora sin memoria virtual, la dirección virtual se coloca directamente en el bus de memoria y hace que se lea o escriba la palabra de memoria física que tiene la misma dirección. Cuando se usa memoria virtual, las direcciones virtuales no pasan directamente al bus de memoria; en lugar de ello, se envían a una **unidad de administración de memoria** (**MMU**), un chip o colección de chips que transforma las direcciones virtuales en direcciones de memoria física (ver figura).

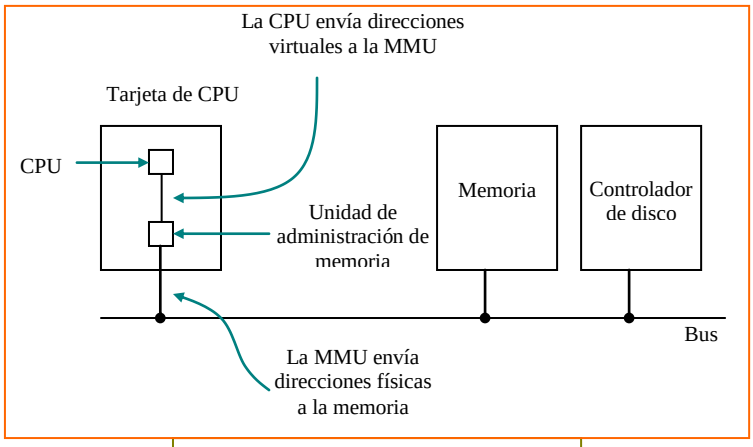
Así, una dirección tal como:

MOVE REG, 1000

se transformará en:

MOVE REG, XXXXX

donde XXXXX es la dirección física de la dirección virtual 1000.



Posición y función de la MMU

Vistazo

Los algoritmos de administración de memoria vistos son necesarios por requerimientos básicos. Las instrucciones que están siendo ejecutadas deben estar en la memoria física. La primer aproximación para encontrar éste requerimiento es la de ubicar el espacio de direcciones lógicas completo en la memoria física. De hecho, una examinación a programas reales muestran que, en muchos casos, el programa entero no es necesitado. Por ejemplo:

- Los programas a menudo tienen código para manejar condiciones de errores inusuales. Ya que estos errores raras veces ocurren, si es que ocurren, el código casi nunca es ejecutado.
- A los arreglos, listas, y tablas a menudo se les asigna mucha más memoria de la que necesitan. Un arreglo puede ser declarado de 100 * 100 elementos, aun aunque rara vez sea más grande que 10 * 10 elementos. Una tabla de símbolos asembler puede tener lugar para 3000 símbolos, aunque el programa tenga menos que 200 símbolos.
- Ciertas opciones o características de un programa rara vez son usadas.

Aun en aquellos casos donde el programa entero es necesitado, puede que no todo sea necesitado en un mismo tiempo.

La habilidad de ejecutar un programa que esta parcialmente en memoria puede tener los siguientes beneficios:

- Un programa no podría ser restringido en tamaño por la cantidad de memoria disponible que hay. Los usuarios podrían ser capaces de escribir programas de un espacio de direcciones virtuales muy grande, simplificando la tarea de programar.
- Ya que cada programa de usuario podría tomar menos memoria física, más programas podrían estar corriendo a la vez, con un correspondiente incremento de la utilización de la CPU y throughput, pero sin un incremento en el tiempo de respuesta o en el turnaround.
- Menos I/O podría ser necesitada para cargar o cambiar cada programa de usuario en la memoria, por lo que cada programa correría más rápido.

Así, correr un programa que no esta completamente en memoria beneficiaria tanto al usuario como al sistema.

La memoria virtual es la separación de la memoria lógica del usuario y la memoria física. Esta separación permite tener una memoria virtual extremadamente grande cuando solo se dispone de una pequeña cantidad de memoria física (Figura 9.1).

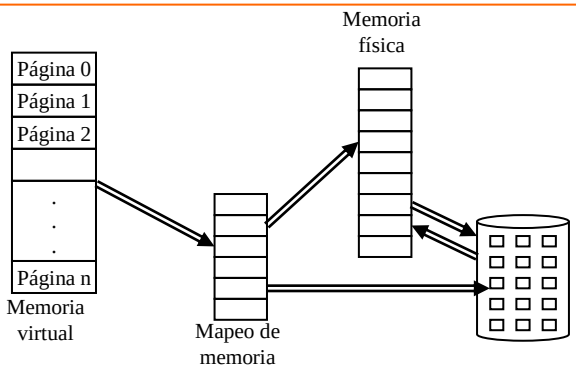


Figura 9.1 Diagrama que muestra la memoria virtual más grande que la memoria física.

La memoria virtual comúnmente es implementada por paginación a demanda, aunque también puede ser implementada en sistemas de segmentación. Varios sistemas proveen un esquema de segmentación paginado, donde los segmentos son divididos en páginas. Segmentación a demanda también puede ser otra de las maneras de implementar la memoria virtual. Sin embargo, los algoritmos de reemplazo de los segmentos son más complejos que los algoritmos de reemplazo de páginas ya que los segmentos tienen tamaños variables.

Paginado por demanda

Un sistema de paginado bajo demanda es similar a un sistema de paginado con swapping (Figura 9.2). Los procesos residen en memoria secundaria (el cual es usualmente un disco). Cuando queremos ejecutar un proceso, lo traemos a la memoria (swap in). Pero en realidad no se trae el proceso completo sino aquellas páginas que se necesiten (es decir, que se demanden, que se pidan). Como ahora los procesos se ven como un conjunto de páginas y no como una única unidad, ya no utilizaremos el término swapper sino pager, el cual se relaciona con las páginas de un proceso (swap se relaciona con procesos completos). Ante esto, se utilizara el término pager en relación con el paginado bajo demanda.

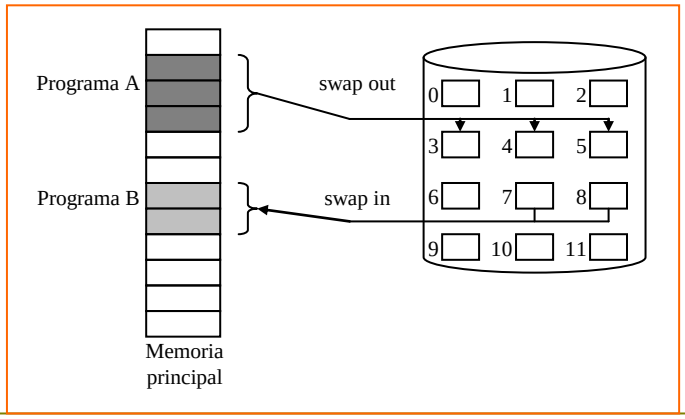


Figura 9.2 Transferencia de una memoria paginada a lugares contiguos de memoria.

Cuando un proceso está para ser traído a memoria, el pager adivina cuales son las páginas que serán usadas antes de que el proceso sea nuevamente sacado de la memoria. En lugar de traer el proceso completo, lo que hace el pager es traer a memoria aquellas páginas que se necesitaran. Esto evita traer a memoria páginas que nunca se usaran, disminuyendo el tiempo de swap y la cantidad de memoria física necesitada.
Con este esquema, se necesita soporte de hardware para saber cuales son las páginas que están en memoria de aquellas que no lo están. Ante esto, se puede usar el bit de valido-invalido. Sin embargo, esta vez cuando el bit esta seteado a valido indica tanto que la página es legal y además esta en memoria. Si el bit esta seteado a invalido indica o que la página no es valida (es decir, no es del espacio de direcciones lógicas del proceso), o es valida pero la página se encuentra en disco. La entrada de la tabla de páginas para una página que esta siendo traída a memoria es seteada como de costumbre, mientras que la entrada de la tabla de páginas para una página que no esta actualmente en memoria es simplemente marcada como invalida, o contiene la dirección de la página en el disco (Figura 9.3).

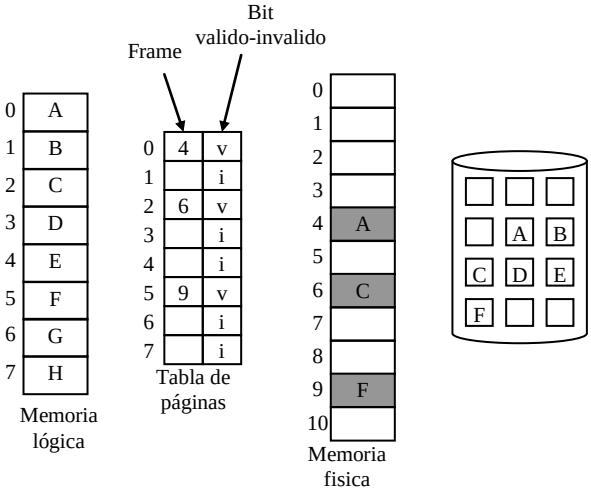


Figura 9.3 Tabla de página cuando algunas páginas no están en memoria principal.

Note que marcar una página como invalida no tendrá efecto si el proceso nunca intenta acceder a la misma. De hecho, si el pager adivinó todas las páginas que usará el proceso mientras ejecuta, entonces nunca se verá una página que tenga el bit en inválido. Pero en caso de que el proceso intente acceder a una página que no esta en memoria, el bit marcado como invalido provocara un trap al sistema operativo por fallo de página. Los pasos cuando se intenta acceder a una página que no esta en memoria son los siguientes (Figura 9.4):

1. Chequear en una tabla interna (normalmente mantenida por el PCB) de este proceso, para determinar si fue una referencia de acceso a memoria valida o invalida.
2. Si la referencia fue invalida, terminar el proceso. Si fue valida, pero la página no esta en memoria, se traerá la página.
3. Se encuentra un frame libre (tomando uno de la lista de frames libres, por ejemplo).
4. Traer la página del disco y ubicarla en el frame elegido.
5. Cuando se completo la operación de lectura, modificar la tabla interna para ese proceso y la tabla de página para indicar que la página ahora esta en memoria.
6. Reiniciar la instrucción que fue interrumpida por el trap de dirección ilegal. Ahora, el proceso puede acceder a la página como si siempre hubiera estado en memoria.

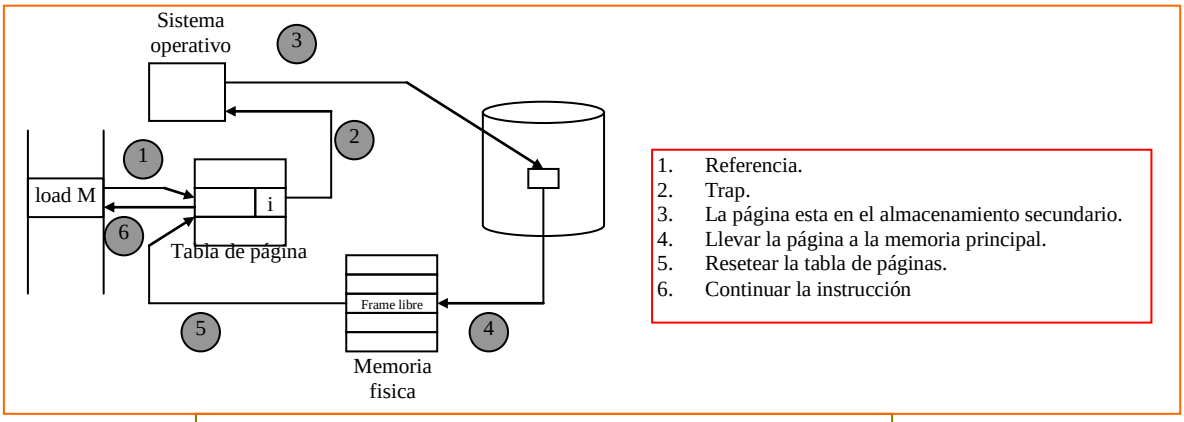


Figura 9.4 Pasos en el manejo de un fallo de página.

Note que como se almacena el estado del proceso interrumpido (registros, código de condición, contador de instrucción) cuando ocurre un fallo de página, podemos continuar el proceso exactamente en el mismo lugar y estado, excepto el hecho de que la página deseada está ahora en memoria y se puede acceder. De esta forma, somos capaces de ejecutar un proceso, aun aunque haya porciones que no estén en memoria. Cuando el proceso trata de acceder a lugares que no están en memoria, el hardware envía un trap al sistema operativo (fallo de página). El sistema operativo lee la página deseada, la ubica en memoria y continúa el proceso como si la página siempre hubiese estado en memoria.
En el caso extremo, el proceso se comenzaría a ejecutar sin páginas en memoria. Cuando el sistema operativo setea el puntero de instrucción con la primer instrucción del proceso, la cual pertenece a una página que no esta en memoria, el proceso provocara inmediatamente un fallo de página. Luego de que esta página fue traída a memoria, el proceso continuara su ejecución fallando hasta que cada página que necesite este en memoria. En este punto, el proceso ejecutara sin fallos. Este esquema es el paginado bajo demanda puro: nunca una página es llevada a memoria hasta que no se necesite.
Teóricamente, algunos programas pueden acceder a varias nuevas páginas de memoria con la ejecución de cada instrucción (una página para la instrucción y muchas para los datos), posiblemente causando muchos fallos de página por instrucción. Esta situación provocaría una caída en la performance, pero generalmente las instrucciones hacen referencia a partes locales.
El hardware para soportar paginado bajo demanda es el mismo que el utilizado para soportar paginado y swapping:

- *Tabla de páginas:* esta tabla tiene la habilidad de marcar una entrada invalida a través del bit valido-invalido o por valores especiales de bits de protección.
- *Memoria secundaria:* esta memoria almacena aquellas páginas que no están presentes en memoria principal. La memoria secundaria es usualmente un disco de alta velocidad.

Además de este hardware, se necesita considerable software.
También se deben imponer algunas restricciones en cuanto a la arquitectura. Un problema importante es la necesidad de ser capaz de recomenzar cualquier instrucción luego de un fallo de página. Un fallo de página

puede ocurrir en cualquier referencia a memoria. Si el fallo de página ocurre cuando la instrucción se está trayendo (es decir, se fue a buscar la instrucción a la memoria principal y no estaba), es decir, en la parte fetch (se va a ubicar en la CPU), se puede recomenzar la instrucción simplemente ubicando nuevamente la instrucción en la CPU. Si el fallo ocurre cuando se está buscando un operando, se debe reubicar la instrucción completa en la CPU, decodificarla nuevamente, y luego finalmente ir a buscar el operando (ya que de seguro esta porque el fallo fue provocado por dicho operando).
Como el peor caso, consideremos la instrucción de tres direcciones tal como Add el contenido de A a B y ubicar el resultado en C. Las etapas para ejecutar esta instrucción podrían ser:

1. Buscar y decodificar la instrucción (ADD).
2. Buscar A.
3. Buscar B.
4. Sumar A y B.
5. Almacenar la suma en C.

Si el fallo ocurre cuando el resultado se va a almacenar en C (ya que C esta en una página que no esta en memoria), debemos ir a buscar la página deseada, corregir la tabla de página, y recomenzar la instrucción. Este recomenzado requerirá ir a buscar la instrucción nuevamente, buscar nuevamente los dos operandos, y luego sumar nuevamente.
La mayor dificultad ocurre cuando una instrucción puede modificar varios lugares diferentes. Por ejemplo, consideremos la instrucción MVC del sistema IBM 360/370, el cual puede mover hasta 256 caracteres desde un lugar a otro, posiblemente sobreescribiendo lugares. Si el bloque fuente o destino se sobrescriben, el bloque fuente puede estar siendo modificado, por lo que en tal caso no basta con simplemente recomenzar la instrucción. Una solución a este problema es usar registros temporarios para mantener los valores que se vayan a sobreescribir. En caso de que exista un fallo de página, todos estos valores son escritos en memoria antes de que ocurra el trap al sistema operativo. Esta acción restaura la memoria a su estado anterior de que la instrucción sea iniciada, por lo que la instrucción puede ser repetida.
Un problema de arquitectura similar ocurre en maquinas que usan modos de direccionamiento especial, incluyendo modos de autoincremento u autodecremento (por ejemplo, el PDP-11). Estos modos de direccionamiento usan un registro como un puntero y automáticamente decrementan o incrementan el registro como indicado. El autodecremento automáticamente decrementa el registro *before* usando su contenido como la dirección del operando; autoincremento automáticamente incrementa el registro *after* usando su contenido como la dirección del operando. Así, la instrucción:

MOV (R2)+, -(R3)

copia el contenido del lugar apuntado por el registro 2 en el lugar apuntado por el registro 3. El registro 2 es incrementado (por 2, ya que PDP-11 es una computadora direccionable a byte) luego de que es usado como un puntero; el registro 3 es decrementado (por 2) antes de que es usado como un puntero. Ahora consideremos que ocurre si ocurre un fallo de página cuando tratamos de almacenar en el lugar apuntado por el registro 3. Para recomenzar la instrucción debemos resetear los dos registros a los valores que tenían antes de que comience la ejecución de la instrucción. Una solución es la de crear un nuevo registro de estado especial para grabar el número de registro y la cantidad modificada por cualquier registro que es modificado durante la ejecución de la instrucción. Este registro de estado permitirá al sistema operativo anular los efectos de una instrucción ejecutada parcialmente que causa un fallo de página.
Estos ejemplos muestran algunos de los problemas que puede causar agregar paginación por demanda a una arquitectura existente. La paginación es agregada entre la CPU y la memoria. Así, muchas personas creen que la paginación puede ser agregada a cualquier sistema. Aunque ésta idea es verdadera en un entorno de paginación no a demanda, donde un fallo de página representa un error fatal, esto no es verdadero en el caso donde un fallo de página solo significa que una página adicional debe ser traída a la memoria y el proceso continuara.

Performance del Paginado bajo demanda

El paginado bajo demanda puede tener un importante efecto en el rendimiento del sistema. Para ver porque, analicemos el tiempo de acceso efectivo para una memoria con paginado bajo demanda. El tiempo de acceso a memoria, *ma*, para la mayoría de las computadoras de hoy en día va entre 10 a 200 nanosegundos. Mientras no tengamos fallos de página, el tiempo de acceso efectivo será igual al tiempo de acceso a memoria. Sin embargo, si ocurre un fallo de página, debemos primero leer la página de disco, y luego acceder a la palabra deseada.
Sea *p* la probabilidad de que ocurra un fallo de página ($0 \leq p \leq 1$). Lo que esperamos es que p este cercano a 0, es decir, que solo ocurran unos pocos fallos de página. El tiempo de acceso efectivo será entonces:

Tiempo-de-acceso-efectivo = (1 – p) * ma + p * tiempo-de-fallo-de-página

Para computar el tiempo de acceso efectivo debemos conocer cuanto tiempo se necesita para satisfacer un fallo. Un fallo de página causa que ocurra la siguiente secuencia:

1. Trap al sistema operativo.
2. Almacenar los registros del usuario y el PCB del proceso.
3. Determinar que la interrupción fue por un fallo de página.
4. Chequear que la referencia a la página es legal y determinar el lugar de la página en el disco.
5. Emitir una lectura desde el disco a un frame libre.
    (a) Esperar en una cola para este dispositivo (disco) hasta que el pedido de lectura es servido.
    (b) Esperar por la búsqueda del dispositivo y/o el tiempo de latencia.
    (c) Comenzar la transferencia de la página al frame libre.
6. Mientras se espera, asignar la CPU a algún otro usuario (CPU scheduling. Este paso es opcional, es decir, puede que no saquemos el proceso si es que ocurre un fallo de página).
7. Interrupción desde el disco (I/O se completo).
8. Almacenar los registros y PCB del otro usuario (si el paso 6 se ejecuto).
9. Determinar que la interrupción fue desde el disco.
10. Corregir la tabla de página y otras tablas para que muestren que la página deseada esta ahora en memoria.
11. Esperar que la CPU sea asignada a este proceso nuevamente.
12. Restituir los registros del usuario, estado del proceso, y la nueva tabla de página, luego continuar la instrucción interrumpida.

No todos estos pasos pueden ser necesarios en todos los casos. Por ejemplo, asumimos que en el paso 6, la CPU es asignada a otro proceso mientras ocurre la I/O. Esto permite mantener la multiprogramación, pero requiere tiempo adicional para continuar la rutina de servicio de fallo de página cuando la transferencia de I/O es completada.
En cualquier caso, debemos enfrentar tres componentes importantes en el tiempo de servicio de un fallo de página:

1. Servir la interrupción de fallo de página.
2. Leer en la página.
3. Reiniciar el proceso.

La primera y la tercer tarea pueden ser reducidas, con un buen código, a varias cientas de instrucciones. Estas tareas pueden llevar desde 1 a 100 microsegundos cada una. El tiempo de cambio de página, por otro lado, será probablemente cercano a 24 milisegundos. Un disco duro típico tiene un promedio de latencia de 8 milisegundos, una búsqueda de 15 milisegundos, y un tiempo de transferencia de 1 milisegundo. Así, el tiempo total de paginado será cercano a 25 milisegundos, incluyendo el tiempo de hardware y software. Recuerde que solo estamos averiguando el tiempo de servicio de dispositivo. Si una cola de procesos están

esperando por un dispositivo (otros procesos que han causado un fallo de página), debemos sumar el tiempo de espera en la cola del dispositivo.
Si tomamos un promedio de servicio de respuesta de un fallo de página en 25 milisegundos y un tiempo de acceso a memoria de 100 nanosegundos, entonces el tiempo de acceso a dispositivo en nanosegundos es:

Tiempo de acceso efectivo = (1 – p) * (100) + p * (25 milisegundos)
= (1 – p) * 100 + p * 25.000.000
= 100 + 24.999.900 * p.

Vemos que entonces el tiempo de acceso efectivo es directamente proporcional a la tasa de fallo de página. Si un acceso de cada 1000 causa un fallo de página, el tiempo de acceso efectivo es de 25 microsegundos (100 + 24999900 * 0.001 = 25099.9 nanosegundos = 25 milisegundos). El cálculo será lento por un factor de 250 por el paginado bajo demanda. Si queremos menos del 10% de disminución, necesitamos:

110 > 100 + 25.000.000 * p
10 > 25.000.000 * p
p < 0.0000004.

Es decir, para mantener la caída de la performance, debido al paginado, a un nivel razonable, podemos permitir solo menos de un acceso a memoria por fallo de página de cada 2500000 accesos.
Es importante mantener la tasa de fallo de página bajo en un sistema de paginado bajo demanda. De otra forma, el tiempo de acceso efectivo se incrementa, disminuyendo dramáticamente la ejecución del proceso.

Reemplazo de páginas

La tasa de fallo de página no es un problema serio, ya que cada página fallara como mucho una sola vez, es decir, cuando es referenciada por primera vez. Esta representación no es del todo correcta. Consideremos que, si un proceso de 10 páginas esta usando solo la mitad de ellas, entonces el paginado bajo demanda ahorrara la I/O necesaria para cargar las 5 páginas que nunca serán usadas. También podemos aumentar nuestro grado de multiprogramación corriendo el doble de la cantidad de procesos. Así, si tenemos 40 frames, podremos correr 8 procesos, en vez de correr 4 si cada uno requiere 10 frames (5 nunca son usados).
Si aumentamos nuestro grado de multiprogramación, estamos sobre-asignando la memoria. Si corremos 6 procesos, cada uno de 10 páginas de tamaño, pero actualmente utilizando solo 5 páginas, debemos aumentar la utilización de la CPU y el throughput, con 10 frames sobrantes. Sin embargo, es posible que cada uno de estos procesos puedan de repente tratar de usar las 10 páginas, resultando en una necesidad de 60 frames, cuando solo están disponibles 40. Aunque esta situación puede que sea imposible, se convierte mucho más posible si incrementamos el nivel de multiprogramación, por lo que el promedio de uso de la memoria esta cercana a la memoria física disponible (en nuestro ejemplo, ¿por qué parar el nivel de multiprogramación a 6, cuando podemos aumentar el nivel a 7 u 8?).
La sobre-asignación se puede producir en las siguientes formas. Mientras un proceso de usuario esta ejecutando ocurre un fallo de página. El hardware enviara un trap al sistema operativo, el cual chequeara su tabla interna para ver que ha ocurrido un fallo de página y no un acceso ilegal a memoria. El sistema operativo determina entonces donde esta residente en disco la página deseada, pero luego encuentra que no existen frames libres en la lista de frames libres; toda la memoria esta en uso (Figura 9.5).

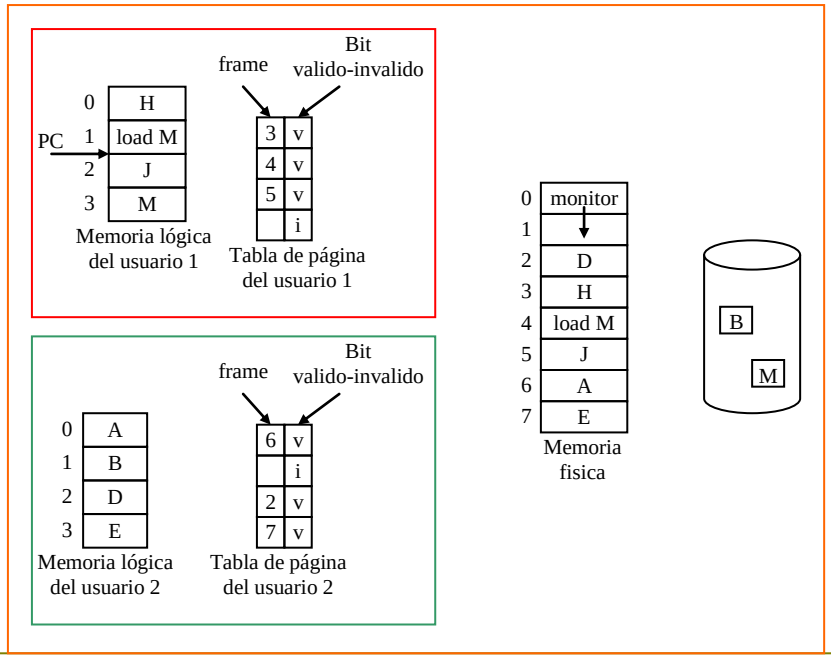


Figura 9.5 Figura que muestra la necesidad del reemplazo de una página.

En este punto el sistema operativo tiene varias opciones. Podría terminar el proceso del usuario. Sin embargo, el paginado bajo demanda es algo que el sistema operativo esta haciendo para mejorar la utilización del sistema y el throughput. El usuario no debe ser consciente de que sus procesos están corriendo en un sistema paginado. El paginado debe ser lógicamente transparente al usuario, por lo que ésta opción no es tomada en cuenta.

Otra opción sería sacar el proceso de ejecución, liberando todos sus frames, y reduciendo el nivel de multiprogramación. Esta opción es una buena idea a veces y se verá más adelante. La opción que veremos en esta sección es la de reemplazo de página.

El reemplazo de página toma los siguientes arreglos. Si no existe frame libre, debemos encontrar uno que actualmente está siendo usado y liberarlo. Podemos liberar un frame por medio de escribir su contenido en disco, y cambiar la tabla de página (y todas las otras tablas) para indicar que la página no esta más en memoria (Figura 9.6).

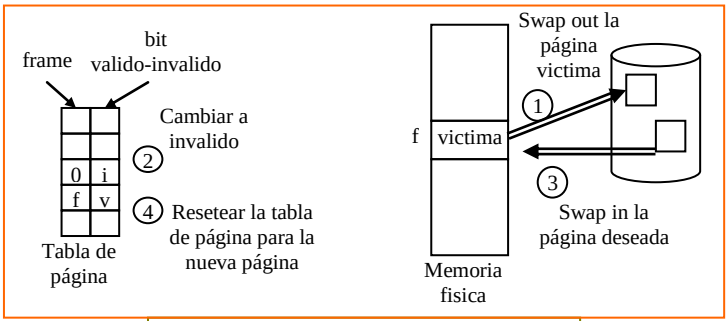


Figura 9.6 Reemplazo de página.

El frame libre puede ahora ser usado para mantener la página por la cual el proceso fallo. La rutina de servicio de fallo de página es ahora modificada para incluir el reemplazo de página:

1. Encontrar el lugar de la página deseada en el disco.
2. Encontrar un frame libre:
   (a) Si hay un frame libre, usarlo.
   (b) De otra manera, usar un algoritmo de reemplazo de página para seleccionar un frame víctima.

(c) Escribir la página víctima en el disco, cambiar las tablas de páginas y de frames.
3. Copiar la página deseada en el frame libre, cambiar las tablas de frames y de página.
4. Continuar el proceso del usuario.

Note que, si no hay frames libres, se requieren dos transferencias de páginas (una para sacar y otra para poner). Esta situación efectivamente dobla el tiempo requerido para servir en un fallo de página e incrementará el tiempo de acceso efectivo.
Este overhead puede ser disminuido por el uso de un bit de modificación (dirty). Cada página o frame tienen un bit asociado de modificación en el hardware. El bit de modificación para una página es seteado (es decir, se pone a uno) por el hardware cuando cualquier palabra o byte en la página es escrito, indicando que la página ha sido modificada. Cuando se selecciona una página para reemplazar, se examina su bit de modificación. Si el bit esta seteado (es decir, esta a uno), sabemos que la página ha sido modificada desde que fue traída desde el disco. En este caso, debemos escribir la página en el disco. Si el bit de modificación no esta seteado, la página no ha sido modificada desde que fue traída a memoria. Ante esto, si la página elegida como víctima no ha sido modificada podemos evitar el hecho de escribir la página en el disco antes de que su frame sea utilizado. Este esquema puede reducir significativamente el tiempo para servir a un fallo de página, ya que se reduce el tiempo de I/O a la mitad si la página no ha sido modificada.
Para implementar el paginado bajo demanda se deben resolver dos problemas principales: debemos desarrollar un algoritmo de asignación de frame y un algoritmo de reemplazo de página. En caso de que tengamos múltiples procesos en memoria, debemos decidir cuantos frames se le asigna a cada proceso. Además, cuando se requiere que una página sea reemplazada, debemos designar un frame que es el que será reemplazado. Diseñar algoritmos apropiados para resolver estos problemas es una tarea importante, ya que la I/O a disco es costosa.

Algoritmos de reemplazo de página

Existen muchos algoritmos diferentes de reemplazo de página. Probablemente cada sistema operativo tiene su propio esquema de reemplazo. Para ilustrar los algoritmos de reemplazo de página usaremos el string de referencia:

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

para una memoria con tres frames.

Algoritmo FIFO: el algoritmo más simple de reemplazo de página es el algoritmo FIFO. Un algoritmo FIFO asocia con cada página el tiempo cuando la página fue traída a memoria. Cuando una página debe ser reemplazada se elige la página más vieja. Note que no es estrictamente necesario almacenar cuando la página fue traída a memoria. Podemos crear una cola FIFO para mantener todas las páginas en memoria. Se reemplaza entonces la página de la cabeza de la cola. Cuando una página es traída a memoria se inserta en la raíz de la cola.
Para nuestro ejemplo de string de referencia, nuestros tres frames están inicialmente vacíos. Las primeras tres referencias (7, 0, 1) causan fallos de página y son traídas a los frames libres. La siguiente referencia (2) es reemplazada por la página 7, ya que 7 fue la que primero se trajo. Ya que 0 es la siguiente referencia y 0 ya esta en memoria, no existe fallo en esta referencia. La primer referencia a la página 3 resulta en el reemplazo de la página 0, ya que ésta es la primera de las tres páginas traídas que existen en memoria (0, 1 y 2). Este reemplazo también provoca que nuestra siguiente referencia a la página 0 provoque un fallo de página. La página 1 es entonces reemplazada por la página 0. Este proceso continúa como se ve en la figura 9.8. La cantidad total de fallos es de 15.

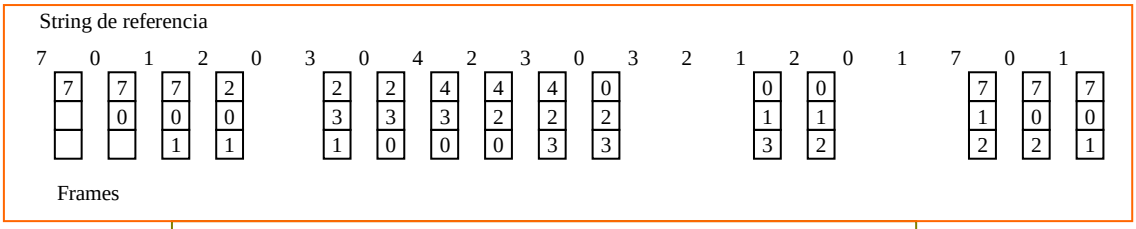


Figura 9.8 Algoritmo de reemplazo de página FIFO.

El rendimiento de este algoritmo no es bueno. La página que se reemplace puede ser un módulo de inicialización que fue usado hace poco y que no volverá a ser usado en un futuro próximo, o puede que la página reemplazada contiene una constante que es muy utilizada y esta en constante uso.
Además, puede ocurrir que se produzca la anomalía de Belady, el cual es el hecho de que si para una cantidad de frames disponibles ocurren n fallos, se supone que si aumentamos el número de frames la cantidad de fallos disminuya. Pues esto no siempre es así y es conocida como la anomalía de Belady, el cual para algunos algoritmos de reemplazo, la cantidad de fallos puede incrementar si se incrementa el número de frames disponibles.

Algoritmo optimal: Un algoritmo de reemplazo de página optimal tiene la taza de fallos de página más baja de todos los algoritmos. Este algoritmo nunca sufrirá la anomalía de Belady y lo que hace es reemplazar la página que no será usada por el periodo más largo de tiempo.
Este algoritmo garantiza que si se utiliza se producirá la menos cantidad de fallos de página. Por ejemplo, para nuestro string de referencia, el algoritmo optimo de reemplazo de página producirá 9 fallos, como se ve en la figura 9.10.

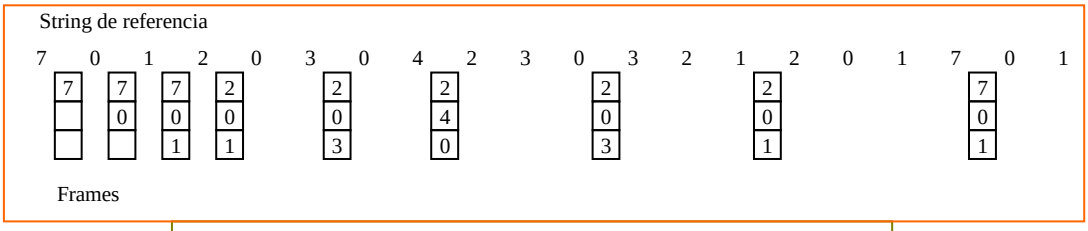


Figura 9.10 Algoritmo de reemplazo de página optimal.

Las primeras tres referencias causan fallos de páginas llenando los tres frames vacíos. La referencia a la página 2 es reemplazada por la página 7, ya que 7 no será usada hasta la referencia 18, mientras que la página 0 será usada en la referencia 5, y la página 1 en la 14. La referencia a la página 3 es reemplazada por la página 1, ya que la página 1 será la ultima de las tres páginas que están en memoria a ser referenciada otra vez. Notemos que solo hay 9 fallos de página el cual es mucho más bajo si lo comparamos con el algoritmo FIFO que tuvo 15 fallos (si ignoramos los primeros tres fallos que todos los algoritmos deben sufrirlos, el algoritmo optimo es el doble de bueno que el algoritmo FIFO).
Desafortunadamente, el algoritmo de reemplazo optimo es difícil de implementar, ya que requiere un conocimiento a futuro de las referencias a memoria (similar a la situación con el algoritmo de scheduling de CPU SJF). Como resultado, este algoritmo es utilizado principalmente para estudios de comparación. Por ejemplo, puede ser bastante útil conocer que, aunque un nuevo algoritmo no es optimo, éste está a 12.3% del optimo en el peor caso, y en 4.7% en el caso promedio.

Algoritmo LRU: si el algoritmo optimo no es posible, quizá una aproximación de éste sea posible. Ante esto, lo que se hará es usar el pasado reciente como una aproximación al futuro cercano, entonces reemplazaremos la página que no ha sido usada por el periodo más largo de tiempo (Figura 9.11). Este algoritmo es el LRU: least recently used.

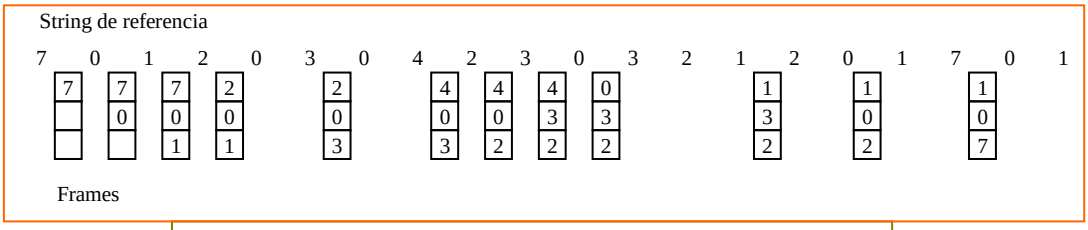


Figura 9.11 Algoritmo de reemplazo de página LRU.

Este algoritmo asocia a cada página el tiempo de su último uso. Cuando se debe reemplazar una página, LRU elige la página que no ha sido usada por el periodo de tiempo más largo. Esta estrategia mira hacia atrás en el tiempo, mientras que el algoritmo optimo miraba hacia delante. Aplicar LRU a nuestro string de referencia produce 12 fallos.

El mayor problema de este algoritmo es como implementarlo. Implementar el algoritmo de reemplazo de página LRU puede requerir bastante soporte de hardware. El problema es determinar el orden de los frames según su último uso. En este punto existen dos implementaciones:

- Contadores: a cada entrada de la tabla de página se le asocia un campo de tiempo de uso, y en cada referencia a memoria, a la página que se referencia se le copia el valor del reloj en dicho campo, con lo que por medio de este método siempre se tiene el tiempo que se hizo referencia por última vez a cada página. Se reemplaza la página con el valor de tiempo más pequeño. Este esquema requiere una búsqueda en la tabla de página para encontrar la página LRU, y una escritura a memoria (para completar el campo de tiempo de uso en la tabla de página) para cada acceso a memoria.
- Stack: otra forma de implementar el algoritmo LRU es mantener una pila de números de páginas. Cuando se hace una referencia a una página, esta es removida de la pila y es ingresada en el tope. De esta forma, en el tope de la pila están las páginas usadas más recientemente y en el fondo esta la página LRU (Figura 9.12).

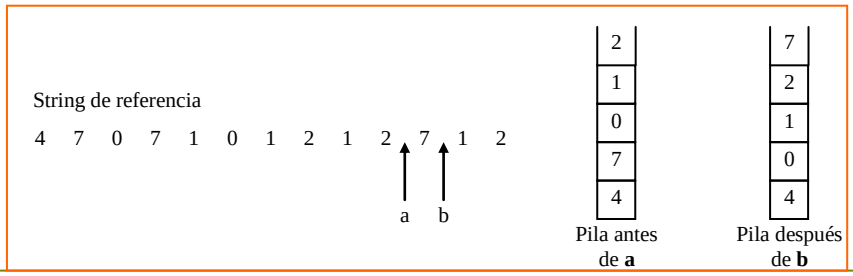


Figura 9.12 Uso de una pila para grabar las referencias de páginas más recientes.

Ya que las entradas deben ser movidas desde el medio de la pila, es mejor implementar este algoritmo con una lista doblemente vinculada, con un puntero a la cabeza y otro a la raíz. Ante esto, mover una página y ponerla en el tope de la pila requerirá cambiar seis punteros en el peor caso.

Los algoritmos optimo y LRU no sufren de la anomalía de Belady.

Note que ninguna implementación del LRU podría ser concebible sin la asistencia del hardware más allá de los registros estándar TLB. El cambio del campo de tiempo de uso o de la pila deben ser realizados en cada referencia a memoria. Si llegamos a usar una interrupción para cada referencia, para permitir que el software cambie dichas estructuras de datos, esto haría lenta cada referencia a memoria, produciendo por consiguiente que sea más lenta la ejecución del proceso del usuario. Pocos son los sistemas que toleran esto.

Algoritmos de aproximación del LRU: Pocos son los sistemas que proveen suficiente hardware para soportar el verdadero reemplazo de página LRU. Algunos sistemas no proveen soporte de hardware por lo

que otros algoritmos de reemplazo de página deben ser usados (FIFO). Sin embargo, algunos sistemas proveen ayuda en forma de un bit de referencia. El bit de referencia para una página es seteado, por el hardware, cuando se hace una referencia a la página (ya sea para una lectura o una escritura en cualquier byte de la página). Los bits de referencia son asociados con cada entrada en la tabla de página.
Inicialmente, todos los bits son puestos en 0 por el sistema operativo. Cuando se ejecuta un proceso del usuario, el bit asociada con cada página referenciada es seteado (es decir, puesto a 1) por el hardware. Luego de algún tiempo, podemos determinar cuales páginas han sido usadas y cuales no examinando el bit de referencia. No conocemos el orden de uso, pero conocemos cuales han sido usadas y cuales no. Este orden de información parcial deja lugar a la existencia de muchos algoritmos de reemplazo que aproximan al LRU.

- *Algoritmo con bit de referencia adicional:* podemos ganar información adicional del orden almacenando los bit de referencia a intervalos regulares. Podemos mantener 8 bits para cada página en una tabla en memoria. A intervalos regulares (digamos cada 100 milisegundos), un timer interrumpe y transfiere el control al sistema operativo. El sistema operativo, para cada página que esta en memoria en ese periodo, hace un shift a la derecha del registro de referencia agregando el bit de referencia de esa página como el nuevo bit de mayor orden y perdiendo el bit de menor orden. Este registro de 8 bits mantiene la historia del uso de la página en los últimos 8 periodos de tiempo. Si un registro contiene 00000000, entonces la página no ha sido utilizada en los últimos 8 periodos de tiempo. Si en los últimos 8 periodos una página fue usada al menos una vez en cada periodo, entonces el registro contendrá el registro 11111111.
  Una página con un registro de historia de 11000100 (196 en decimal) ha sido usada más recientemente que una con el registro 01110111 (119 en decimal). Si interpretamos estos 8 bits como un entero sin signo, la página con el número más bajo es la página LRU, y debe ser reemplazada. Note que no se garantiza que este número sea único. Ante esto, si existen varias páginas LRU se puede aplicar un mecanismo de elección sobre ellas tal como FIFO.
  El número de bits de este registro de historia puede ser variable. En el caso extremo, el número puede ser reducido a 0, dejando solo el bit de referencia. Este algoritmo es llamado algoritmo de reemplazo de segunda chance.
- *Algoritmo de reemplazo de segunda chance:* El algoritmo de reemplazo de segunda chance es básicamente un algoritmo del tipo FIFO. Sin embargo, cuando una página debe ser seleccionada se inspecciona su bit de referencia. Si el valor es 0, se procede a reemplazar esta página. Si el valor es 1, se deja a esta página una segunda chance y se mueve a ver si la siguiente página puede ser seleccionada. Cuando una página obtiene su segunda chance, su bit de referencia es limpiado (puesto en 0) y su tiempo de llegada es reestablecido al tiempo actual. Así, a una página que se le da la segunda chance no será reemplazada hasta que todas las demás páginas sean reemplazadas (o se les otorgue la segunda chance). Además, si una página es usada muy seguido manteniendo así su bit de referencia a 1, nunca será reemplazada. Si la página es muy usada se mantiene su bit a uno ya que cada vez que se hace una referencia a la misma se pone el bit de referencia a uno.
  Una forma de implementar la segunda chance es la de tener una cola circular. Un puntero indica que página será la próxima a ser reemplazada. Cuando se necesita un frame, el puntero avanza hasta que encuentra una página con el bit de referencia a 0. A medida que va avanzando, limpia los bits de referencia de las demás páginas (es decir, si va avanzando y encuentra que las páginas tienen su bit de referencia a 1, los pone en 0 logrando así la segunda chance) (Figura 9.13). Una vez que se encuentra una víctima, la página es reemplazada y la nueva página es insertada en la cola circular en la posición que se fue la anterior. Note que, en el peor caso, cuando todos los bits están seteados (en 1), el puntero dará una vuelta completa a través de toda la lista, otorgando a cada página una segunda oportunidad.

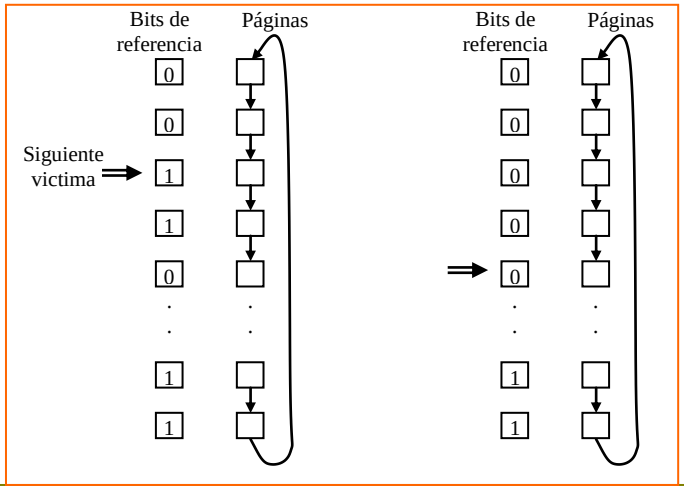


Figura 9.13 Algoritmo de reemplazo de página de segunda chance.

- *Algoritmo de segunda chance mejorado:* el algoritmo de segunda chance puede ser mejorado si consideramos el bit de referencia y el bit de modificación como un par ordenado. Con estos dos bits, tenemos los siguientes cuatro posibles casos:
    1. (0, 0) ni recientemente usado ni modificado, mejor página para reemplazar.
    2. (0, 1) no recientemente usado pero modificado, no es bastante bueno, ya que la página debe ser escrita en disco antes de ser reemplazada.
    3. (1, 0) recientemente usada pero sin modificar, probablemente será usada nuevamente pronto.
    4. (1, 1) recientemente usada y modificada, probablemente será usada otra vez, y debe ser escrita en disco en caso de ser elegida para ser reemplazada.

  Cuando se llama al procedimiento de reemplazo de página, cada página esta en uno de éstos cuatro estados. Lo que se hace es tener una lista circular con las páginas y en el momento de elegir una página se examina cada página para ver a cual clase pertenece y se reemplaza la primer página que se encuentra teniendo el estado (0, 0). Note que quizá debemos buscar varias veces en la cola circular antes de encontrar la página víctima.
  Este método es usado en el esquema de administración de memoria de Macintosh.

Algoritmos de conteo: Existen muchos otros algoritmos que pueden ser usados para el reemplazo de página. Por ejemplo, podríamos mantener un contador del número de referencias que han sido hechas a cada página, y desarrollar los siguientes dos esquemas:

- Algoritmo LFU: el algoritmo de reemplazo de página LFU (least frequently used) requiere que la página con la menor cantidad de referencias sea reemplazada. La razón para esta elección es que una página que es muy usada tendrá un número muy grande. Este algoritmo tiene el problema de que páginas que al principio son muy usadas tendrán en un momento un gran número, pero puede que luego la página no se use más, pero como el gran uso del inicio provoco un número grande, la página quedara en memoria. Una solución es ir disminuyendo su número gradualmente.
- Algoritmo MFU: el algoritmo de reemplazo de página MFU (most frequently used) esta basado en el argumento de que la página con la cantidad más baja fue traída a memoria hace poco y puede que todavía tenga uso.

Ninguno de los dos algoritmos anteriores es común. Sus implementaciones son bastante caras y no se aproximan al algoritmo optimo.

Algoritmo de buffering de página: Otros procedimientos son a menudo usados a demás de los algoritmos de reemplazo de página. Por ejemplo, algunos sistemas normalmente mantienen una pileta de frames libres.

Cuando ocurre un fallo de página, un frame víctima se elige con el algoritmo. Sin embargo, la página es leída en un frame libre de los de la pileta antes de que el frame víctima es copiado a disco. Este procedimiento permite al proceso reiniciar tan pronto como sea posible, sin esperar que el frame víctima deba ser copiado a disco. Una vez que el frame víctima es copiado a disco, éste frame es agregado a la pileta de frames libres.
Una extensión de esta idea es la de mantener una lista de páginas modificadas. Cuando el dispositivo de paginado esta ocioso, una página es seleccionada y copiada a disco. Ante esto, su bit de modificación es entonces reseteado. Este esquema incrementa la probabilidad de que una página este sin modificar cuando es seleccionada para ser reemplazada, y no se necesitara copiarla nuevamente a disco.
Otra modificación es la de mantener una pileta de frames libres, pero recordando que páginas estuvieron en cada frame. Ya que el contenido del frame no es modificado cuando este es escrito en disco, la vieja página puede ser reusada directamente desde la pileta de frames libres si es que se necesita antes de que dicho frame sea reusado. En este caso no se necesita I/O. Al ocurrir un fallo de página, primero se chequea si la página deseada está en la pileta de frames libres. Si no esta, se debe seleccionar un frame de los libres y copiarla en él.
Este esquema es usado en los sistemas VAX/VMS, con un algoritmo de reemplazo FIFO. Cuando el algoritmo FIFO equivocadamente reemplaza una página que esta todavía en uso activo, la página es rápidamente recuperada del buffer de frames libres, y así no se necesita I/O. El buffer de frames libres provee protección contra la relativa pobreza del algoritmo FIFO.

Asignación de frames

Ahora el problema se centra principalmente en cuantos frames deben ser otorgados a cada proceso. El caso más simple de memoria virtual es el del sistema de un único usuario. Consideremos un sistema de un único usuario con 128K de memoria compuestas de páginas de 1K. Ante esto, existen 128 frames. El sistema operativo puede tomar 35K, dejando 93 frames para los procesos del usuario. Bajo el paginado bajo demanda puro, los 93 frames serian inicialmente puestos en la lista de frames libres. Cuando un proceso del usuario comienza su ejecución, éste generara una secuencia de fallos de página. Los primeros 93 fallos de página tendrán un frame libre de la lista de frames libres. Cuando la lista de frames libres se agota, un algoritmo de reemplazo de página será usado para elegir una de los 93 páginas en memoria para ser reemplazada. Cuando el proceso termina, los 93 frames podrían ser ubicados nuevamente en la lista de frames libres.
Existen muchas variaciones de esta estrategia simple. Podemos requerir que el sistema operativo aloje todo el espacio de tablas y espacio de buffer de la lista de frames libres. Cuando éste espacio no esta en uso por el sistema operativo, éste puede ser usado para soportar el paginado del usuario. Podríamos tratar de mantener tres frames libres en la lista de frames libres en todo momento. Así, cuando ocurre un fallo de página, existen frames libres disponibles para que los utilice la página deseada. Mientras toma lugar la etapa de traer la página, puede ser seleccionada una página para ser escrita en disco, el cual se escribirá mientas el proceso del usuario este en ejecución.
Un problema diferente surge cuando el paginado bajo demanda es combinado con la multiprogramación. La multiprogramación pone dos (o más) procesos en memoria al mismo tiempo.

Mínimo número de frames: Por supuesto que hay varias restricciones con nuestras estrategias de asignación de frames. No podemos asignar más del total de frames disponibles (a menos que se compartan páginas). Existe además un mínimo número de frames que pueden ser asignados. Obviamente, a medida que el número de frames asignado a cada proceso decrece, se incrementa la tasa de fallos de página, retrasando la ejecución del proceso.
Hay también un mínimo número de frames que deben ser asignados. Este número mínimo esta definido por el conjunto de instrucciones de la arquitectura. Recuerde que, cuando ocurre un fallo de página antes de que la instrucción fuese totalmente ejecutada, la instrucción debe ser reiniciada. Ante esto, debemos

tener bastantes frames para mantener todas las páginas diferentes que cualquier única instrucción puede hacer referencia.
Por ejemplo, consideremos una maquina en el cual todas las instrucciones de referencia a memoria tienen solo una dirección de memoria. Así, necesitamos al menos un frame para la instrucción y un frame para la referencia a memoria. Además, si se permite un nivel de direccionamiento indirecto (por ejemplo, una instrucción load en la página 16 puede referir a una dirección en la página 0, el cual es una referencia indirecta a la página 23), entonces se requieren al menos tres frames por proceso.
El mínimo número de frames por proceso esta definido por la arquitectura, mientras que el máximo número esta definido por la cantidad de memoria física disponible.

Algoritmos de Asignación: la forma más fácil de dividir *m* frames entre *n* procesos es dar a cada uno una cantidad de *m/n* frames. Por ejemplo, si hay 93 frames y 5 procesos, cada uno obtendrá 18 frames. Los sobrantes 3 frames podrían ser usados como una pileta de frames libres. Este esquema es llamado *asignación igualitaria*.
Una alternativa es reconocer para cada proceso la cantidad de memoria que necesitara. Si un pequeño proceso de estudiante de 10K y una base de datos de 127K son los únicos dos procesos corriendo en el sistema con 62 frames libres, no tiene mucha coherencia dar 31 frames a cada uno, ya que el proceso de estudiante no necesitara más de 10 frames, por lo que 21 frames serán gastados inútilmente.
Para resolver este problema, podemos usar una *asignación proporcional*. Asignaremos la memoria disponible a cada proceso de acuerdo a su tamaño. Si s_i es el tamaño de la memoria virtual para el proceso p_i, definimos:

$$S = \sum s_i.$$

Entonces, si el número total de frames disponibles es m, asignaremos a_i frames al proceso p_i, donde a_i es aproximadamente:

$$a_i = s_i/S*m$$

Por supuesto, debemos ajustar cada a_i a un entero, el cual debe ser mayor que el mínimo número de frames requerido para el conjunto de instrucciones, con una cantidad no mayor a m.
Para la asignación proporcional, podríamos dividir 62 frames entre dos procesos, uno de 10 páginas y uno de 127 páginas, asignando 4 frames y 57 frames respectivamente, ya que:

$$10/137 * 62 \approx 4$$
$$1277 * 62 \approx 57.$$

De esta forma, ambos procesos comparten los frames disponibles de acuerdo a sus "necesidades", en lugar de la misma cantidad para cada uno.
En ambos modos (igualitaria o proporcional), la asignación a cada proceso puede variar de acuerdo al nivel de multiprogramación. Si el nivel de multiprogramación aumenta, cada proceso perderá algunos frames para proveer de memoria a los nuevos procesos. Por otro lado, si el nivel de multiprogramación decrece, los frames que han sido asignados a los procesos salientes pueden ser repartidos entre los procesos que quedan
Note que, con la asignación proporcional o igualitaria, un proceso de alta prioridad es tratado de igual manera que uno de baja prioridad. Sin embargo, por definición debemos desear que aquellos procesos de alta prioridad tengan más memoria para acelerar su ejecución.
La tendencia es usar asignación proporcional donde la razón de frames depende no del tamaño del proceso, sino de su prioridad, o una combinación de ambas (tamaño y prioridad).

Asignación global versus local: otro factor importante en la forma en que los frames son asignados a los procesos es el reemplazo de página. Con múltiples procesos compitiendo por frames, podemos clasificar los algoritmos de reemplazo de página en dos extensas categorías: reemplazo global y reemplazo local. El reemplazo global permite a un proceso seleccionar un frame a ser reemplazado del conjunto de todos los frames, aun si el frame está actualmente asignado a otro proceso; un proceso puede tomar frames de otro proceso.
Por ejemplo, consideremos un esquema de asignación donde permitimos que un proceso de alta prioridad seleccione frames de los procesos de baja prioridad para ser reemplazados. Un proceso puede elegir un frame a reemplazar de entre sus propios frames o de los frames de los procesos de baja prioridad.
Con una estrategia de reemplazo local, el número de frames asignado a un proceso no cambia. Con el reemplazo global, un proceso podía elegir para reemplazar cualquier frame que estuviese asignado a otro proceso, aumentando así el número de frames que se le fue otorgado.

Thrashing

Si el número de frames asignados a un proceso de baja prioridad cae por debajo del número mínimo requerido por la arquitectura de la computadora, se debe suspender la ejecución del proceso, debiendo liberar las páginas que el proceso usaba.
Aunque es técnicamente posible reducir el número de frames asignados a un nivel mínimo, hay un número de páginas que deben estar en uso activo todas juntas. Si el proceso no tiene este número de frames, éste tendrá un fallo de página. En este punto, éste debe reemplazar alguna página. Como todas las páginas están en uso activo, reemplazara una página que necesitara rápidamente. Consecuentemente, fallara rápidamente por la página que le falta, y así otra vez. El proceso continuara fallando, reemplazando páginas por la cual más tarde fallara, sin poder avanzar en su ejecución.
Esta alta actividad de reemplazo de página es llamado *thrashing*. Un proceso esta en thrashing si gasta más tiempo en el paginado que en su ejecución.

Causa del thrashing: El thrashing puede provocar serios problemas de performance. Consideremos el siguiente caso, el cual esta basado en la actual conducta de los antiguos sistemas de paginado.
El sistema operativo monitorea la utilización de la CPU. Si la utilización de la CPU es demasiado baja, se incrementara el grado de multiprogramación ingresando nuevos procesos al sistema. Si se usa un algoritmo de reemplazo global de páginas, las páginas serán reemplazadas sin considerar al proceso al cual pertenecen. Supongamos que un proceso entra en una fase de ejecución y necesita más frames. Este fallara y tomara páginas de otros procesos. Sin embargo, éstos procesos necesitarán éstas páginas, por lo que fallarán, tomando páginas de otros procesos. Estos procesos que fallan deben usar el dispositivo de paginado para sacar y meter las páginas a memoria (swap in-out). A medida que aumenta la cola del dispositivo de paginado, la cola de listos quedara vacía. Como los procesos están esperando en el dispositivo de paginado, la utilización de la CPU decrece.
El scheduler de la CPU ve este decaimiento, e incrementa el grado de multiprogramación. Estos nuevos procesos tratan de tomar páginas de los procesos que están corriendo (es decir, esperando en el dispositivo de paginado) provocando cada vez más fallos de páginas (es decir, para poder tomar las páginas deben causar primero un fallo), causando que la cola del dispositivo aumente cada vez más su tamaño. Como resultado, la utilización de la CPU cae aun más, y el scheduler de la CPU trata de incrementar el grado de multiprogramación aun más. Un thrashing ha ocurrido. La razón de fallos de página aumenta notoriamente. Como resultado, el tiempo de acceso efectivo a memoria se incrementa. No se hace trabajo alguno, ya que los procesos están gastando su tiempo en el paginado.
Este fenómeno se ve en la figura 9.14. La utilización de la CPU se ve a partir de su grado de multiprogramación. A medida que el grado de multiprogramación aumenta, la utilización de la CPU también aumenta, aunque más lentamente, hasta que se alcanza un máximo. Si el grado de multiprogramación se aumenta aun más, el thrashing provoca que la utilización de la CPU caiga ásperamente. En este punto, para

incrementar el grado de multiprogramación y frenar el thrashing debemos decrementar el grado de multiprogramación.

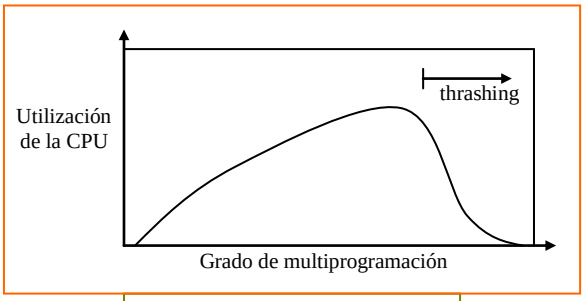


Figura 9.14 Thrashing.

Los efectos del thrashing pueden ser limitados usando un algoritmo de reemplazo local (o prioritario). Con el reemplazo local, si un proceso comienza a estar en estado de thrashing, éste no puede robar frames a otros procesos provocando que ellos en el futuro también caigan en thrashing. Las páginas son reemplazadas con respecto al proceso al cual pertenecen. Sin embargo, si un proceso esta en thrashing, éste estará en la cola del dispositivo de paginado la mayor parte de su tiempo. El tiempo promedio para el servicio de un fallo de página aumentara, debido al tamaño promedio de la cola del dispositivo de paginado. Así, el tiempo de acceso efectivo se incrementara aun para aquellos procesos que no están en thrashing.
Para prevenir el thrashing, debemos proveerle a un proceso la cantidad de frames que necesite. Para saber cuantos frames necesitara un proceso existen varias técnicas. La técnica de *working-set* (que se verá más adelante) lo logra mirando que cantidad de frames un proceso esta usando en un momento. Esta estrategia define el modelo de lugar de un proceso en ejecución.
El modelo de lugar declara que, mientras un proceso esta en ejecución, este se mueve de un lugar a otro. Un lugar es un conjunto de páginas que están usadas activamente todas juntas. Un programa esta generalmente compuesto de varios diferentes lugares, el cual pueden solaparse.
Por ejemplo, cuando se llama a una subrutina, ésta define un nuevo lugar. En este lugar, las referencias a memoria están hechas a las instrucciones de la subrutina, a sus variables locales, y a un subconjunto de variables globales. Al terminarse la subrutina, el proceso abandona su lugar ya que las variables locales y las instrucciones de la subrutina ya no están en uso activo. Podemos retornar a este lugar más tarde. Así, vemos que los lugares están definidos por la estructura del programa y sus estructuras de datos.
Supongamos que asignamos bastantes frames a un proceso para alojar su actual lugar. Este fallara para las páginas en su lugar hasta que éstas páginas estén en memoria; luego, no fallara otra vez hasta que cambie de lugar. Si asignamos menos frames que el tamaño del lugar actual, el proceso entrara en thrashing, ya que no puede mantener en memoria todas las páginas que esta activamente usando.

Modelo Working-Set: el modelo working-set esta basado en la suposición de lugar. Este modelo usa un parámetro Δ, que define la ventana de working-set. La idea es la de examinar las más recientes Δ páginas referenciadas. El conjunto de páginas en las más recientes Δ páginas referenciadas es el working-set (Figura 9.16). Si una página esta en uso activo, ésta estará en el working-set. Si hace tiempo que no es usada, ésta caerá fuera de la ventana Δ unidades de tiempo después de su ultima referencia. Así, el working-set es una aproximación del lugar del programa.
Por ejemplo, dando la secuencia de referencias a memoria que se ven en la figura 9.16, si Δ = 10 referencias a memoria, entonces el working-set en t_1 es {1, 2, 5, 6, 7}. Para el t_2, el working-set ha cambiado a {3, 4}.

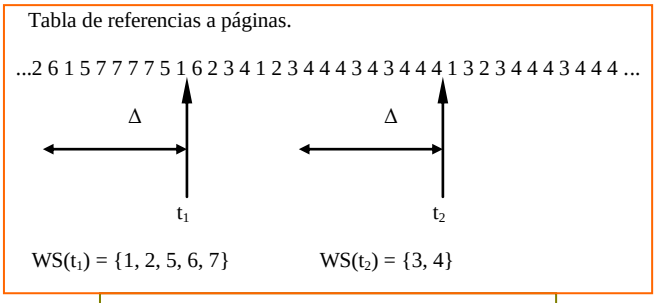


Figura 9.16 Modelo working-set.

La precisión del working-set depende de la selección del Δ. Si Δ es demasiado chico, éste no abarcara el lugar completo. Si Δ es demasiado grande, éste solapara varios lugares. En el caso extremo, si Δ es infinito, el working-set es el conjunto de las páginas alcanzadas durante la ejecución del proceso.
La propiedad más importante del working-set es su tamaño. Si calculamos el tamaño del working-set, WSS_i, para cada proceso en el sistema, podemos entonces considerar:

$$D = \Sigma\ WSS_i,$$

donde D es la demanda total de frames. Cada proceso esta activamente usando las páginas en su working-set. Así, el proceso i necesita WSS_i frames. Si el total demandado es mayor que el número total de frames disponibles (D > m), ocurrirá un thrashing, ya que algunos procesos no tendrán suficientes frames.
El uso del modelo del working-set es entonces bastante simple. El sistema operativo monitorea el working-set de cada proceso y asigna a éste working-set los suficientes frames para proveerle a éste con su tamaño de working-set. Si existen bastantes frames extras, otro proceso puede ser iniciado. Si la suma de los tamaños de working-set se incrementa, excediendo el número total de frames disponibles, el sistema operativo debe seleccionar un proceso y suspenderlo. Las páginas del proceso son escritas en disco y sus frames son reasignados a los demás procesos. El proceso suspendido puede ser continuado más tarde.
Esta estrategia de working-set previene el thrashing mientras mantiene el grado de multiprogramación tan alto como sea posible. Ante esto, se optimiza la utilización de la CPU.
La dificultad con el modelo de working-set es mantener la pista de working-set. La ventana del working-set es una ventana en movimiento. En cada referencia a memoria, una nueva referencia aparece de un lado de la ventana y la referencia más vieja se cae del otro lado de la ventana. Una página está en el working-set si fue referenciada en cualquier parte en la ventana de working set. Podemos aproximar el modelo de working-set con una interrupción a intervalos fijos de un timer y un bit de referencia.
Por ejemplo, asumimos que Δ es 10.000 y podemos causar una interrupción de timer cada 5.000 referencias. Cuando ocurre una interrupción del timer, se copia y se limpian los valores del bit de referencia para cada página. Así, si ocurre un fallo de página, podemos examinar el actual bit de referencia y dos bits en memoria para determinar si una página fue usada entre las últimas 10.000 a 15.000 referencias. Si ésta fue usada, al menos uno de esos bits debe estar en 1. Si la página no ha sido usada, estos bits estarán en 0. Aquellas páginas que tengan por lo menos un bit en 1 formaran parte del working-set. Note que el arreglo no tiene mucha precisión, ya que no podemos decir donde ocurrió la referencia en un intervalo de 5.000. Podemos disminuir la incertidumbre incrementando el número de nuestros bits de historia y el número de interrupciones (por ejemplo, 10 bits e interrupciones cada 1000 referencias). Sin embargo, el costo para servir éstas interrupciones más frecuentes será alto también.

Frecuencia de fallos de página: el modelo working-set es exitoso y el conocimiento del working-set puede ser muy útil para prepaging (se verá más adelante), pero parece ser una torpe forma de controlar el thrashing. La estrategia de frecuencia de fallo de página (PFF) toma una idea más directa.
El problema especifico es como prevenir el thrashing. El thrashing tiene una alta tasa de fallo de página. Así, lo que podemos controlar es la tasa de fallo de página. Cuando ésta tasa es demasiado alta, sabremos

que el proceso necesita más frames. Similarmente, si es demasiado baja, entonces el proceso puede tener demasiados frames. Podemos establecer limites superiores e inferiores para una tasa de fallo de página deseada. Si la actual tasa de fallo de página excede el límite superior, se asignara al proceso otro frame (es decir, al proceso que tiene una alta tasa de fallo de página se le otorgará un nuevo frame). Si la tasa de fallo de página esta por debajo del limite inferior, se le sacara un frame al proceso. Así, podemos medir y controlar directamente la tasa de fallo de página para prevenir el thrashing.
Como con la estrategia de working-set, podemos tener que suspender un proceso. Si la tasa de fallo de página se incrementa y no existe frame libre, se debe seleccionar un proceso y suspenderlo. Los frames liberados son entonces distribuidos entre los procesos con una alta tasa de fallo de página.

Otras consideraciones

La selección de un algoritmo de reemplazo y la política de asignación son decisiones muy importantes para formar un buen sistema de paginado. Pero también existen otras consideraciones importantes.

Prepaging: una propiedad obvia de un sistema de paginado bajo demanda puro es el gran número de fallos de página que ocurren cuando un proceso comienza su ejecución. Esta situación es un resultado de tratar de conseguir el lugar (locality) inicial en memoria. Lo mismo ocurre otras veces. Por ejemplo, cuando un proceso que estaba suspendido en disco continúa su ejecución, todas sus páginas están en disco y cada una será traída a memoria por su correspondiente fallo de página. Prepaging es un intento de prevenir este alto nivel de paginado inicial. La estrategia es traer a memoria todas las páginas que sean necesarias de una sola vez.
Por ejemplo, en un sistema donde se usa el modelo working-set, se mantiene con cada proceso una lista con las páginas que están en su working-set. En caso de que se deba suspender un proceso (debido a una espera de I/O o a una falta de frames libres), se recuerda el working-set para ese proceso. Cuando el proceso es continuado (por la completitud de la I/O o existen suficientes frames libres), automáticamente se trae el working-set completo a memoria antes de que el proceso se continúe.
Prepaging puede ser ventajoso en algunos casos. La cuestión es simplemente si el costo de hacer prepaging es menor que el costo de servir a los correspondientes fallos de página. Este puede ser el caso de que muchas de las páginas llevadas a memoria por el prepaging no estén en uso.
Si asumimos que *s* páginas son traídas a memoria por el sistema de prepaging, y una fracción *a* de esas *s* páginas están actualmente en uso ($0 \leq a \leq 1$). La cuestión es si el costo de las *a* fallos de página salvados es mayor o menor que el costo de traer por medio de prepaging las $(1 - a)$ páginas. Si *a* es cercano a 0, prepaging no conviene; si *a* es cercano a 1, prepaging conviene.

Tamaño de página: Los diseñadores de un sistema operativo para una maquina existente rara vez tienen una elección en cuanto al tamaño de página. Sin embargo, cuando se está diseñando una nueva maquina, se debe hacer una decisión sobre cual es el mejor tamaño de página. *Pero no existe el mejor tamaño de página.* Más bien, existe un conjunto de factores que soportan varios tamaños. Los tamaños de las páginas son potencias de 2, generalmente entre el rango 512 (2^9) a 16.348 (2^{14}) bytes.
En el momento de seleccionar el tamaño de la página se debe tener en cuenta el tamaño de la tabla de páginas correspondiente. Para un espacio de memoria virtual dado, decrementar el tamaño de la página provocara que se incremente el número de páginas y, por lo tanto, el tamaño de la tabla de páginas. Para una memoria virtual de unos 4 megabytes (2^{22}), habrá 4096 páginas de 1024 bytes cada una pero solo 512 páginas de 8192 bytes. Ya que cada proceso activo debe tener su propia copia de su tabla de páginas, en este caso es deseable tener un tamaño de página grande.
Por otro lado, la memoria es más fácil utilizarla con páginas de tamaño pequeño. Si a un proceso se le asigna memoria desde la posición 00000, continuando hasta donde necesite memoria, el proceso probablemente no terminara exactamente en el límite de la página. Así, una parte de la última página debe ser asignada (ya que las páginas son unidades de memoria) pero no es usada (fragmentación interna). Asumiendo independencia de tamaños de procesos y tamaños de páginas, esperamos que, en el caso

promedio, la mitad de la última página de cada proceso sea gastada. Esta perdida podría ser de solo 256 bytes para una página de 512 bytes, o de 4096 bytes para una página de 8192 bytes. Para minimizar la fragmentación interna, se necesitan páginas de poco tamaño.
Otro problema es el tiempo requerido para leer o escribir una página. El tiempo de I/O esta compuesto por la búsqueda, latencia, y el tiempo de transferencia. El tiempo de transferencia es proporcional a la cantidad transferida (es decir, al tamaño de página), con lo que parecería que es mejor páginas de poco tamaño. Sin embargo, recuerde que la latencia y el tiempo de búsqueda normalmente empequeñecen el tiempo de transferencia. A una tasa de transferencia de 2 megabytes por segundo, tomara solo 0.2 milisegundos transferir una página de 512 bytes. Por otro lado, la latencia es quizá 8 milisegundos y el tiempo de búsqueda es de 20 milisegundos. Por consiguiente, del tiempo total de I/O (28.2 milisegundos) solo el 1 por ciento es atribuible a la actual trasferencia. Doblando el tamaño de la página incrementa el tiempo de I/O a solo 28.4 milisegundos. Esto toma 28.4 milisegundos para leer una página de 1024 bytes de tamaño, pero 56.4 milisegundos leer la misma cantidad como dos páginas de 512 bytes de tamaño cada una. Así, para minimizar el tiempo de I/O se debe aumentar el tamaño de la página.
Con páginas de poco tamaño, sin embargo, el total de I/O debería ser reducido, ya que el lugar (locality) será mejorado. Por ejemplo, consideremos un proceso de 200K de tamaño, el cual solo la mitad (100K) esta siendo usada para la ejecución. Si solo tenemos una gran página, se debe traer la página completa (un total de 200K). Si solo tenemos páginas de 1 byte, entonces podríamos traer solo los 100K que están siendo usados. Con un pequeño tamaño de página, se tiene mejor resolución, permitiendo separar la memoria que esta siendo usada actualmente. Con páginas de gran tamaño, se debe asignar y transferir no solo lo que esta siendo necesitado, sino también cualquier otra cosa que aparezca en la página, ya sea que se necesite o no. Así, una página de poco tamaño resultara en menos I/O y menos memoria total asignada.
Por otro lado, se debe tener en cuenta que con páginas de 1 byte de tamaño se tendrá un fallo de página para cada byte. Un proceso de 200K, usando solo la mitad de la memoria, generara solo un fallo de página para páginas de tamaño de 200K, pero 102.400 fallos de página para páginas de 1 byte de tamaño. Cada fallo de página generara una gran cantidad de overhead necesario para procesar la interrupción, almacenar los registros, reemplazar una página, hacer cola en el dispositivo de paginado, y cambios en las tablas. Para minimizar el número de los fallos de página, se necesitan tener páginas de gran tamaño.
La tendencia es optar por páginas de gran tamaño. De hecho, la primer edición de este libro (1983) usaba 4096 bytes como el límite superior de tamaño de páginas, y este valor fue el más común en el tamaño de página de 1990. El Intel 80386 tiene tamaño de páginas de 4K; el Motorola 68030 permite páginas de tamaño que varían desde 256 bytes a 32K. La evolución de tamaño de página cada vez mayores es producto de la evolución en la velocidad de la CPU y la capacidad de la memoria, incrementándose más rápido que la velocidad de los discos. Los fallos de página son más costosos hoy en día que antes, en el rendimiento global del sistema. Ante esto, es mejor aumentar el tamaño de las páginas para evitar que se produzcan fallos de página. Por supuesto, como resultado esta la fragmentación interna.
Existen otros factores por considerar (tal como la relación entre el tamaño de la página y el tamaño del sector en el dispositivo de paginado). El problema no tiene la mejor respuesta. Algunos factores (fragmentación interna, lugar) sostienen que deben existir páginas de poco tamaño, mientras que otros (tamaño de la tabla de páginas, tiempo I/O) desean páginas de gran tamaño.

Estructura del programa: la paginación bajo demanda esta diseñada para que sea transparente al usuario. En muchos casos, el usuario no sabe como es que se administra la memoria. En otros, el rendimiento del sistema puede ser mejorado si el usuario tiene conocimientos de cómo se administra la memoria.
Como un buen ejemplo, asumamos que tenemos páginas de 128 palabras. Consideremos un programa en Pascal, el cual es una función que solo llena de ceros una matriz de 128 * 128. El código seria:

```
var A: array[128] of array[128] of integer;
        for j := 1 to 128 do
                for i := 1 to 128 do
                        A[i][j] := 0;
```

Note que el arreglo es almacenado por filas, es decir, es almacenado A[1][1], A[1][2], ..., A[1][128], A[2][1], A[2][2], ..., A[2][128], ..., A[128][128]. Para páginas de 128 palabras, cada fila toma una página. Así, el procedimiento codifica un 0 en una página, luego un 0 en otra, luego otro 0 en otra, y así. Si el sistema operativo asigna menos de 128 frames al programa entero, entonces su ejecución tendrá 128 * 128 = 16.384 fallos de página. Si se cambia el código anterior por el siguiente:

```
var A: array[128] of array[128] of integer;
        for j := 1 to 128 do
                for i := 1 to 128 do
                        A[j][i] := 0;
```

se llenará de ceros todos los elementos de una página antes de pasar a la siguiente, reduciendo el número de fallos de página a 128.

Si se tiene cuidado en las estructuras de datos y en las estructuras del programa se puede incrementar la localidad y así bajar la tasa de fallos de página y el conjunto de páginas en el working-set. Una pila tiene buena localidad ya que los accesos son siempre hechos en el tope. Por otro lado, la tabla de hash esta diseñada para dispersar las referencias, produciendo una mala localidad. Por supuesto, la localidad de referencias es solo una de las medidas de eficiencia del uso de una estructura de datos. Otros factores importantes son la velocidad de búsqueda, el número total de referencias a memoria, y el número total de páginas alcanzadas.

Un paso más adelante, el compilador y el cargador pueden tener un efecto importante sobre el paginado. La separación del código y los datos generando código reentrante significa que las páginas de código pueden ser solo de lectura y así, nunca serán modificadas. Las páginas que no son modificadas no tienen que ser llevadas a disco en el momento de ser reemplazadas. El cargador puede evitar poner rutinas en los limites de las páginas, manteniendo cada rutina completamente en una página. Las rutinas que se llaman entre sí varias veces pueden ser empaquetadas en una misma página.

También puede afectar la elección de un lenguaje de programación. Por ejemplo, LISP usa frecuentemente punteros, y los punteros tienden a aleatorizar los accesos a memoria. En contraposición, Pascal usa pocos punteros. Los programas escritos en Pascal tienen mejor localidad de referencia y por lo tanto generalmente se ejecutaran más rápido que los escritos en LISP en un sistema de memoria virtual.

I/O Interlock: Cuando se usa la paginación bajo demanda, a veces se necesita que algunas de las páginas estén bloqueadas en memoria. Una de tales situaciones ocurre cuando se realiza I/O a o desde la memoria (virtual) del usuario. La I/O es a menudo implementada por un procesador de I/O independiente. Por ejemplo, a un controlador de disco magnético se le da generalmente el número de bytes a transferir y una dirección de memoria para el buffer (Figura 9.18). Al completarse la transferencia, la CPU es interrumpida.

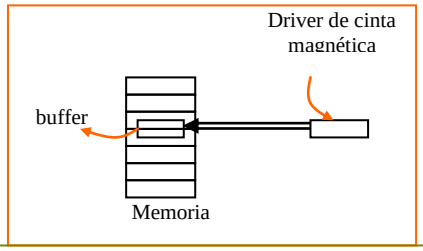


Figura 9.18 Diagrama que muestra porque los frames usados por la I/O deben estar en memoria.

Se debe tener seguridad de que la siguiente secuencia de eventos no ocurre. Un proceso emite un pedido de I/O, y es puesto en la cola para el dispositivo de I/O. Mientras tanto, la CPU es otorgada a otro proceso. Estos procesos causan fallos de página y, usando un algoritmo de reemplazo de página global, uno de estos fallos reemplaza la página conteniendo el buffer de memoria para el proceso que esta esperando por la I/O. Ante esto, las páginas son sacadas de la memoria. Un tiempo más tarde, cuando el pedido de

I/O del proceso avanza hasta la cabeza de la cola del dispositivo, la I/O ocurre a la dirección especificada. Sin embargo, el frame esta ahora siendo usado por una página diferente perteneciente a otro proceso.
Existen dos soluciones posibles a este problema. Una solución es nunca ejecutar una I/O a la memoria del usuario. En lugar de esto, los datos son siempre copiados entre la memoria del sistema y la memoria del usuario. La I/O toma lugar solo entre la memoria del sistema y el dispositivo de I/O. Para escribir un bloque en cinta, primero se copia el bloque en la memoria del sistema, y luego se escribe en la cinta. Esta copia extra puede ser inaceptable por su alto overhead. Otra solución es la de permitir que las páginas sean bloqueadas en memoria. Un bit de bloqueo es asociado con cada frame. Si el frame esta bloqueado, éste no puede ser seleccionado para ser reemplazado. Bajo esta restricción, para escribir un bloque en cinta, primero se bloquean todas las páginas de memoria conteniendo el bloque. Al completarse la I/O se desbloquean las páginas.
Otro uso del bit de bloqueo involucra el normal reemplazo de páginas. Consideremos la siguiente secuencia de eventos. Un proceso de baja prioridad produce un fallo de página. Seleccionando un frame para ser reemplazado, el sistema de paginado lee la página necesaria en memoria. Listo para continuar, el proceso de baja prioridad entra en la cola de listos y espera por la CPU. Ya que éste es un proceso de baja prioridad, puede que no sea seleccionado por el scheduler de la CPU por algún largo tiempo. Mientras este proceso de baja prioridad esta esperando, un proceso de alta prioridad falla. En busca de alguna página para reemplazar, el sistema de paginado ve que hay una página que no esta siendo referenciada y que además no esta modificada: la página del proceso de baja prioridad que esta esperando en la cola de listos y que dicha página recién fue traída a memoria. Esta página se ve como la candidata perfecta para ser reemplazada, ya que esta limpia y además no esta siendo usada.
El decidir si el proceso de alta prioridad puede ser capaz de reemplazar las páginas de los procesos de baja prioridad es una decisión de política. Luego de esto, si es así, estamos dando prioridad a los procesos de alta prioridad sobre los procesos de baja prioridad. Por otro lado, estamos gastando el esfuerzo gastado para traer la página del proceso de baja prioridad. Si decidimos prevenir el reemplazo de una página que recién ha sido traída a la memoria hasta que puede ser usada por lo menos una vez, entonces podemos usar el bit de bloqueo para implementar este mecanismo.
Sin embargo, usar el bit de bloqueo puede ser peligroso en caso de que éste sea puesto en 1 pero nunca es vuelto a cero. Tal situación podría ocurrir (por ejemplo, debido a un fallo del sistema operativo) provocando que los frames bloqueados no se puedan usar más. El sistema operativo de Macintosh provee un mecanismo de bloque de páginas, ya que éste es un sistema de único usuario, y el sobreuso del bloqueo dañaría solo al usuario que lo hace. Pero en sistema multi usuarios se necesita tener menos confianza. Por ejemplo, SunOS permite bloqueos indirectos, ya que es libre de olvidar los bloqueos en caso de que la pileta de frames libres este muy vacía.

Segmentación por demanda

Aunque la paginación por demanda es considerada el sistema de memoria virtual más eficiente, se requiere una significativa cantidad de hardware para implementarlo. Cuando no se tiene dicho hardware, una alternativa es la segmentación por demanda. El Intel 80286 no proveía características de paginación, pero tenia segmentos. El sistema operativo OS/2, el cual corría en tal CPU, usa el hardware de segmentación para implementar segmentación bajo demanda como el único acercamiento a la paginación bajo demanda.
El sistema OS/2 asignaba la memoria en segmentos, en vez de en páginas. El sistema mantenía pista de los segmentos a través del descriptor de segmentos, el cual incluía información sobre el tamaño del segmento, protecciones, y ubicación. Un proceso no necesitaba tener todos sus segmentos en memoria para ejecutar. En lugar de esto, el descriptor de segmentos contenía un bit de validez para cada segmento que indicaba si el segmento estaba actualmente en memoria. Cuando un proceso direccionaba un segmento, el hardware chequeaba su bit de validez. Si el segmento estaba en memoria, el acceso continuaba sin impedimento. En caso de que el segmento no estuviera en memoria, ocurría un trap al sistema operativo (fallo de segmento), así como en las implementaciones de paginado bajo demanda. Luego, OS/2 sacaba un segmento de la

memoria y lo llevaba a disco, y traía el segmento pedido entero. Así, continuaba la instrucción interrumpida (la que produzco el fallo).
Para determinar cual era el segmento a reemplazar en caso de que se produzca un fallo, OS/2 usa otro bit en el descriptor de segmentos llamado bit de acceso. Un bit de acceso servia de la misma manera como el bit de referencia en la paginación bajo demanda. Este bit era seteado siempre que cualquier bit en el segmento fuera leído o escrito. Se mantiene también una cola conteniendo una entrada para cada segmento en memoria. Luego de cada cierto tiempo, el sistema operativo ubicaba en la cabeza de la cola cualquier segmento que tuviera dicho bit de acceso seteado. Luego limpiaba todos los bits de accesos. Ante esto, la cola siempre quedaba ordenada con los segmentos más recientemente usados en la cabeza. Además, OS/2 proveía llamadas a sistema que los procesos pueden usar para informar al sistema sobre aquellos segmentos que, o bien pueden ser eliminados, o bien deben permanecer en memoria. Esta información se usaba para reorganizar las entradas de la cola. Al ocurrir un trap de segmento invalido, la rutina de administración de memoria primero analiza si existe suficiente espacio de memoria libre para poder alojar al nuevo segmento. En este momento se puede realizar la compactación para eliminar la fragmentación externa. Si, luego de la compactación, no existe todavía suficiente espacio libre, se realiza el reemplazo de alguno de los segmentos. El segmento en el final de la cola es elegido para ser reemplazado y es escrito en disco para obtener espacio. Si este nuevo espacio es suficiente para alojar al segmento pedido, entonces dicho segmento es leído, se modifica el descriptor de segmentos, y el segmento es ubicado en la cabeza de la cola. De otra manera, se realiza nuevamente la compactación de la memoria y el procedimiento comienza nuevamente.
Esta simple y clara forma de manejar la segmentación por demanda requiere bastante overhead.

10. Interface File-System

Para la mayoría de los usuarios, el sistema de archivos es la parte más visible de un sistema operativo. Provee el mecanismo para almacenar y acceder tanto a los datos y a los programas del sistema operativo y a todos los usuarios del sistema. El sistema de archivos consiste de dos partes diferentes: una colección de *archivos*, una unión de datos relacionados; y una estructura de *directorio*, el cual organiza y provee información sobre todos los archivos que están en el sistema. Algunos sistemas de archivos tienen una tercer parte, las *particiones*, las cuales son usadas para separar lógica o físicamente grandes colecciones de directorios. En este capítulo se verán los aspectos de los archivos, y las variedades de las estructuras de directorio. También se verá la forma de manejar la protección de archivos, el cual es necesaria en entornos donde múltiples usuarios tienen acceso a los archivos, y donde es deseable controlar por quien y de que formas se tuvo acceso al archivo.

Concepto de archivo

Las computadoras pueden almacenar información en varios medios diferentes, tales como discos magnéticos, cintas magnéticas, y discos ópticos. Para que al sistema le sea conveniente usar este sistema de archivos, el sistema operativo provee una vista uniforme de ésta información almacenada. El sistema operativo abstrae las propiedades físicas de los dispositivos de almacenamiento definiendo una unidad de almacenamiento lógico, el archivo. Los archivos son mapeados, por el sistema operativo, en dispositivos físicos. Estos dispositivos de almacenamiento son usualmente no volátiles, por lo que su contenido persiste aunque haya un fallo de energía y se deba reiniciar el sistema.

Un archivo es una colección de información relacionada que se graba sobre almacenamiento secundario. Comúnmente, los archivos representan programas (tanto de la forma fuente como de la forma objeto) y datos. Los archivos de datos pueden ser numéricos, alfabéticos, alfanuméricos, o binarios. Los archivos pueden ser de forma libre tales como los archivos de texto, o pueden estar rígidamente formateados. En general, un archivo es una secuencia de bits, bytes o líneas, el cual su significado esta definido por el creador de archivo y el usuario.

La información de un archivo esta definida por el creador. Muchos tipos diferentes de información se pueden almacenar en un archivo: programas fuente, programas objeto, programas ejecutables, datos numéricos, textos, imágenes grafica, sonidos, etc. Un archivo tiene una estructura definida de acuerdo a su tipo. Un archivo de texto es una secuencia de caracteres organizado en líneas (y posiblemente páginas); un archivo fuente es una secuencia de subrutinas y funciones, cada una además organizada como declaraciones seguidas de sentencias ejecutables; un archivo objeto es una secuencia de bytes organizados en bloques comprensibles para el linker del sistema, un archivo ejecutable es una serie de secciones de código que el cargador puede traer a memoria y ejecutar.

Atributos de un archivo: un archivo tiene un nombre y es referenciado por tal nombre. Un nombre es usualmente un string de caracteres. Cuando un archivo recibe un nombre, este se independiza del proceso, del usuario y aun del sistema donde fue creado. Por ejemplo, un usuario puede crear un archivo llamado "example.c", mientras que otro usuario puede modificar dicho archivo por medio de especificar su nombre. El dueño del archivo puede escribir el archivo en un disquete y leerlo en otro sistema, donde aun este puede seguir llamándose "example.c".

Un archivo tiene otros atributos que son:

- *Nombre.*
- *Tipo.*
- *Ubicación.*
- *Tamaño.*
- *Protección.*
- *Hora, día, e identificación del usuario:* esta información puede ser mantenida para la creación, la ultima modificación, y el último uso. Estos datos son muy útiles para la protección, seguridad, y monitoreo del uso.

La información sobre todos los archivos es mantenida en la estructura de directorio que también reside en el almacenamiento secundario. Puede tomar desde 16 a más de 1000 bytes para grabar la información de cada archivo. En un sistema con muchos archivos, el tamaño del directorio puede ser de megabytes. Ya que los directorios, como los archivos, deben ser no volátiles, deben ser almacenados en el dispositivo secundario y traídos a memoria gradualmente a medida que se necesite.

Operaciones sobre los archivos: un archivo es un tipo de dato abstracto. Para definir un archivo, necesitamos considerar las operaciones que se pueden realizar sobre él. El sistema operativo provee llamadas al sistema para crear, escribir, leer, reubicación en el archivo, eliminar, y truncar un archivo. A continuación se verán cuales son las operaciones que debe tomar el sistema operativo para poder realizar cada una de las acciones anteriores:

- *Crear un archivo:* Se necesitan dos pasos para crear un archivo. Primero, se debe encontrar espacio en el sistema de archivos para este nuevo archivo. Segundo, se debe hacer en el directorio una nueva entrada para este nuevo archivo. Las entradas del directorio graban el nombre del archivo y el lugar en el sistema de archivo.
- *Escribir un archivo:* Para escribir un archivo, se hace una llamada al sistema especificando tanto el nombre del archivo, y la información a escribir en el archivo. Dando el nombre del archivo, el sistema busca en el directorio para localizar el archivo. El sistema debe mantener un puntero de escritura para ubicar en el archivo donde será la próxima escritura. El puntero de escritura debe ser cambiado siempre que ocurre una escritura.
- *Leer un archivo:* Para leer desde un archivo, usamos una llamada al sistema que especifica el nombre del archivo y donde (en memoria) será puesto el siguiente bloque de archivo. Una vez más, se debe hacer una búsqueda en el directorio y el sistema debe mantener un puntero de lectura para ubicar en el archivo donde tomara lugar la siguiente lectura. Una vez que la lectura toma lugar, se debe modificar el puntero de lectura. Ya que, en general, un archivo o esta siendo leído o esta siendo escrito, la mayoría de los sistemas mantienen un puntero a la ubicación actual en el archivo. Las operaciones de lectura y escritura usan el mismo puntero, ahorrando lugar y reduciendo la complejidad del sistema.
- *Reubicación en el archivo:* Se busca en el directorio la entrada apropiada y la actual posición del puntero en el archivo se setea al valor indicado. La reubicación en un archivo no necesita ninguna operación de I/O. Esta operación es conocida como *seek*.
- *Eliminar un archivo:* Para eliminar un archivo, se busca en el directorio el nombre del archivo. Habiendo encontrado éste, se libera todo el espacio que utilizaba el archivo (para que pueda ser usado por otros archivos) y se borra la entrada en el directorio.
- *Truncar un archivo:* Existen ocasiones donde el usuario quiere que los atributos de un archivo queden iguales, pero quiere borrar el contenido del archivo. En lugar de forzar al usuario a eliminar el archivo y luego recrearlo, ésta función permite que todos los atributos del archivo queden sin cambiar (excepto por el largo del archivo) pero el largo del archivo es reseteado a cero.

Estas seis operaciones básicas son el mínimo conjunto de operaciones requeridas para archivos. Otras operaciones comunes incluyen agregar nueva información al final de un archivo existente, y renombrar un archivo existente. Estas operaciones primitivas pueden entonces ser combinadas para realizar otras operaciones de archivos. Por ejemplo, crear una copia de un archivo, o copiar el archivo en otro dispositivo de I/O, tal como una impresora o una pantalla, puede ser llevado a cabo por medio de la creación de un nuevo archivo y leyendo desde el archivo vejo y escribiendo en el nuevo. También se desea que haya operaciones que permitan a un usuario obtener los atributos de un archivo y setear alguno de estos atributos. Por ejemplo, podemos desear tener una operación que permita a un usuario determinar el estado de un archivo, tal como su tamaño, y tener una operación que permita al usuario setear los atributos del archivo, tal como el dueño del mismo.

La mayoría de las operaciones de archivos mencionadas involucran la búsqueda en el directorio para la entrada asociada con el nombre del archivo. Para evitar esta constante búsqueda, muchos sistemas primero abrirán el archivo cuando es usado por primera vez. El sistema operativo mantiene una pequeña tabla conteniendo la información sobre todos los archivos abiertos (la tabla de archivos abiertos). Cuando se requiere una operación sobre un archivo, se usa un índice en esta tabla, por lo que no se requiere búsqueda alguna. Cuando un archivo no es usado por un largo tiempo, éste es cerrado por el proceso y el sistema operativo elimina su entrada de la tabla de archivos abiertos.
Algunos sistemas implícitamente abren un archivo cuando se hace la primer referencia. El archivo es automáticamente cerrado cuando el programa que lo abrió termina. La mayoría de los sistemas, sin embargo, requieren que el archivo sea abierto explícitamente por el programador por medio de una llamada al sistema (open) antes de que sea usado. La operación open toma un nombre de un archivo y busca en el directorio, copiando la entrada del directorio en la tabla de archivos abiertos, asumiendo que la protección del archivo permite hacer dicho acceso. La llamada al sistema open típicamente retornara un puntero a la entrada en la tabla de archivos abiertos. Este puntero, no el actual nombre de archivo, es usado en las operaciones de I/O, evitando cualquier búsqueda adicional, y simplificando la interfaz de la llamada al sistema.
La implementación de las operaciones open y close en un entorno multiusuario, tal como UNIX, es más complicada. En tales sistemas, varios usuarios pueden abrir el archivo al mismo tiempo. Típicamente, el sistema operativo usa dos niveles de tablas internas. Hay una tabla por proceso de todos los archivos que el proceso tiene abiertos. Almacenada en esa tabla esta la información con respecto al uso del archivo por el proceso. Por ejemplo, se encuentra el puntero actual del archivo, indicando el lugar en el archivo en el cual se realizara la siguiente lectura o escritura.
Cada entrada en la tabla de archivos del proceso a su vez apunta a una tabla de los archivos abiertos en el sistema (systemwide). Dicha tabla contiene la información del archivo que es independiente del proceso, tal como el lugar del archivo en el disco, días de acceso, y tamaño del archivo. Una vez que ya se abrió un archivo por el proceso, en caso de que otro proceso realice una llamada al sistema *open* con el mismo nombre de archivo, simplemente resultara agregar una nueva entrada en la tabla de archivos abiertos para el proceso, con un nuevo puntero de posición del archivo, y un puntero en la entrada correspondiente de la systemwide. Típicamente, la tabla de archivos abiertos del sistema tiene un contador para cada entrada, el cual indica la cantidad de procesos que tienen abierto dicho archivo. Cada llamada close decrementa dicha cantidad, y cuando el contador llega a cero significa que el archivo ya no esta en uso por lo que la entrada de tal en la tabla systemwide es eliminada. En resumen, existen varias piezas asociadas con cada archivo:

- *Puntero de archivo:* el puntero es único para cada proceso operando en el archivo y, por lo tanto, debe estar separado de los atributos del archivo que están en disco.
- *Contador de archivo abierto:* a medida que los archivos se van cerrando, el sistema debe eliminar la entrada de dicho archivo de la tabla systemwide. Ya que varios procesos pueden abrir el mismo archivo, el sistema debe esperar hasta el último close para poder eliminar la entrada para el archivo en la tabla de archivos abiertos del sistema. Este contador lleva pista del número de opens y closes, y alcanza un cero en el último close. Así, el sistema puede eliminar la entrada.
- *Ubicación en el disco del archivo:* la mayoría de las operaciones que se hacen sobre un archivo necesitan que se modifique el mismo. La información necesaria para ubicar el archivo en disco es mantenida en memoria para evitar tener que leerla desde el disco para cada operación que se haga.

Algunos sistemas operativos proveen un mecanismo para bloquear secciones de archivos abiertos para los accesos de múltiples procesos, para compartir secciones de un archivo entre varios procesos, y aun para mapear secciones de un archivo en memoria en un sistema de memoria virtual. Esta última función es llamada *memory mapping*, y permite que parte del espacio de direcciones virtuales se asocie lógicamente con una sección del archivo. Las escrituras y lecturas a ésta región de memoria son tratadas como lecturas y escrituras en el archivo, simplificando en gran manera el uso del archivo. Cerrar el archivo significa que toda la parte de la memoria que fue mapeada es escrita en disco y se elimina de la memoria virtual del proceso. Múltiples procesos pueden estar autorizados a realizar el mapeo de la misma sección del archivo

en el espacio de memoria virtual de cada proceso, para conseguir compartir los datos. Cualquier escritura que se haga por algún proceso modifica los datos de la memoria virtual y puede ser vista por todos los demás procesos que mapearon la misma sección del archivo. Esta forma de compartir información se implementa de la siguiente manera (recordar capitulo 9): cada parte compartida del mapeo de la memoria virtual de cada proceso apunta a la misma página de la memoria física (la página que mantiene una copia del bloque del disco). Ya que los accesos a la parte compartida deben ser coordinados, los procesos involucrados pueden usar uno de los mecanismos de exclusión mutua que se vieron en el capítulo 6 (Figura 10.1).

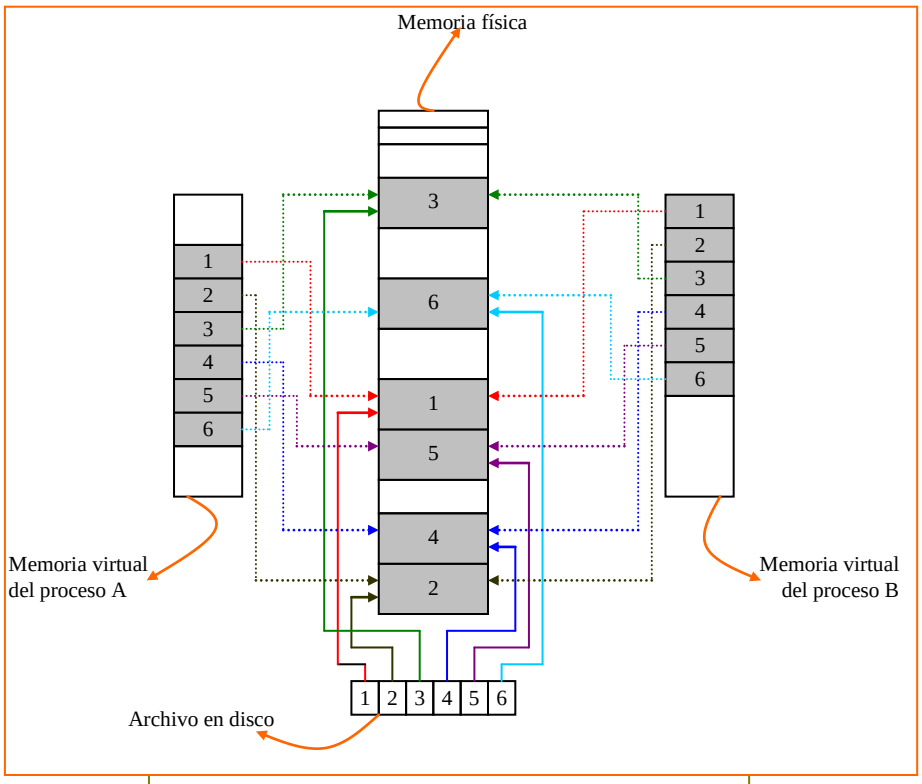


Figura 10.1 Archivos mapeados en memoria.

Tipos de archivos: una consideración importante es si el sistema operativo puede reconocer tipos. Si un sistema operativo puede reconocer de qué tipo es un archivo, entonces puede operar sobre éste de diferentes maneras. Por ejemplo, un error común ocurre cuando un usuario trata de imprimir la forma de un programa en objeto binario. Este intento normalmente produce basura, pero puede ser prevenido si el sistema operativo sabe de que tipo es el archivo.

Una técnica común para implementar los tipos de archivos es que tengan el tipo como parte del nombre. El nombre se divide en dos partes (un nombre y una extensión separado por un punto). Así, un usuario y el sistema operativo pueden decir no solo cual es el nombre del archivo sino también de que tipo es. Por ejemplo, en MS-DOS el nombre consiste de hasta 8 caracteres seguidos de un punto y terminados por una extensión de 3 caracteres. El sistema usa la extensión para conocer el tipo de archivo y el tipo de operaciones que se le pueden aplicar. Por ejemplo, solo las extensiones ".com, ".ese", o ".bat" pueden ser ejecutadas. MS-DOS reconoce poca cantidad de extensiones, pero los programas de aplicación también usan la extensión para indicar los tipos de archivos en el cual están interesados. Por ejemplo, los assemblers esperan tener como entrada un archivo de extensión ".asm", y el procesador de texto WordPerfect espera que sus archivos terminen en ".wp". Las extensiones no se requieren ya que el usuario se puede referir a un archivo sin la extensión y la aplicación buscara un archivo con el nombre dado. Ya que éstas extensiones no son soportadas por el sistema operativo, son consideradas como "indirectas" a las aplicaciones el cual operan sobre ellas.

Otro ejemplo de la utilidad de los tipos de archivos viene del sistema operativo TOPS-20. Si el usuario trata de ejecutar el programa objeto el cual su fuente ha sido modificado desde que el archivo objeto fue

realizado, el archivo fuente será recompilado automáticamente. Esta función asegura que el usuario siempre correrá la versión más actual del archivo objeto. Note que esta opción es posible solo si el sistema operativo discrimina el archivo fuente del archivo objeto, ya que debe chequear la fecha de cada archivo de la última modificación o creación, y determinar el lenguaje del programa fuente (para elegir el compilador correcto).
Consideremos el sistema operativo Macintosh. En este sistema, cada archivo tiene un tipo, tal como "text" o "pict". Cada archivo tiene un atributo creador el cual contiene el nombre del programa que lo creo. Este atributo es seteado por el sistema operativo durante la llamada de creación. Por ejemplo, un archivo producido por un procesador de texto tiene el nombre del procesador de texto como su creador. Cuando el usuario lo abre, haciendo doble clic del mouse en el icono, el procesador de texto es invocado automáticamente, y el archivo se carga, listo para ser editado.
El sistema UNIX es incapaz de proveer tales características ya que utiliza un número mágico almacenado en el comienzo de algún archivo que indica el tipo de archivo: programa ejecutable, archivo batch, etc. No todos los archivos tienen este número mágico, por lo que las características del sistema no se pueden basar solo en este tipo de información. UNIX no registra el nombre del creador del programa.

Estructura del archivo: Los tipos de archivos también pueden ser usados para indicar la estructura interna del archivo. Los archivos objeto y fuente tiene estructuras que coinciden con la forma que esperan los programas que leen estos tipos de archivos. Además, algunos archivos se deben ajustar a ciertas estructuras que es conocida por el sistema operativo. Por ejemplo, el sistema operativo puede requerir que un archivo ejecutable tenga una estructura especifica para que pueda determinar donde cargarlo en memoria y donde se encuentra la primer instrucción.
Aparece la primer desventaja de que un sistema operativo soporte varias estructuras de archivos: el tamaño del sistema operativo es incomodo. Si el sistema operativo define cinco tipos diferentes de estructuras, necesita contener el código para soportar estos cinco tipos. Graves problemas pueden surgir de que nuevas aplicaciones requieran información estructurada en formas de que el sistema operativo no soporta.
Por ejemplo, asumamos que el sistema operativo soporta dos tipos de archivos: archivos de texto (compuestos por caracteres ASCII separados por espacios y enters) y archivos binarios ejecutables. Ahora, si nosotros como usuarios queremos crear otra estructura de archivo, el cual no se parece a un archivo de texto ni a un archivo binario, debemos engañar o emplear de mala manera los mecanismos de tipos de archivos del sistema operativo (ya que debemos elegir uno de los dos tipos), o modificar o abandonar nuestro proyecto de nueva estructura de archivo.
Algunos sistemas operativos soportan una pequeña cantidad de estructuras de archivos. Este es el caso de UNIX y MS-DOS. UNIX considera a cada archivo como una secuencia de bytes de 8 bits cada byte; la interpretación de estos bytes no es realizada por el sistema operativo. Este esquema provee máxima flexibilidad pero poco soporte. Cada programa de aplicación debe incluir su propio código para ser interpretado y ubicar el archivo en la estructura apropiada. Sin embargo, todos los sistemas operativos deben soportar al menos una estructura (que es un archivo ejecutable) para que el sistema sea capaz de cargar y correr programas.

Estructura interna del archivo: internamente, ubicar un offset en el archivo puede ser complicado para el sistema operativo. Los sistemas de disco tienen un tamaño de bloque bien definido (registro físico), y todos los bloques son del mismo tamaño. No es seguro que el tamaño del registro físico coincida exactamente con el tamaño del registro lógico. Además, los registros lógicos pueden variar en tamaño. Empaquetar un número de registros lógicos en bloques físicos es una solución común para este problema.
Por ejemplo, UNIX define a todos los archivos para que sean un stream de bytes. Cada byte es individualmente direccionable por su offset desde el inicio (o fin) del archivo. En este caso, el registro lógico es 1 byte. El sistema de archivo automáticamente empaqueta o des-empaqueta los bytes en un bloque de disco físico (digamos, 512 bytes por bloque) como sea necesario.

El tamaño del registro lógico, el tamaño del registro físico, y la técnica de empaquetamiento determinan cuantos registros lógicos habrá en cada bloque físico. El empaquetado puede ser realizado por alguna aplicación del usuario o por el sistema operativo.
En ambos casos, el archivo es considerado como una secuencia de bloques. Todas las funciones de I/O operan en términos de bloques.
Ante el hecho de que el espacio del disco esta asignado por bloques, puede ocurrir que algo de espacio del último bloque quede sin uso. Esto es la fragmentación interna en el disco. A mayor tamaño de bloque, mayor será la fragmentación interna.

Métodos de acceso

Los archivos almacenan información. Al ser usada, ésta información debe estar en la memoria de la computadora. Existen varias formas para acceder a la información que se encuentra en el disco. Algunos sistemas proveen solo un método de acceso. En otros sistemas, como el IBM, soportan varios métodos de acceso, y elegir el mejor método para una aplicación particular es un gran problema de diseño.

Acceso secuencial: el método de acceso más simple es el acceso secuencial. La información en el archivo es procesada en orden, un registro después de otro. Este método es el más común, por ejemplo, editores y compiladores acceden a archivos de esta manera.
Las operaciones en el archivo son leer y escribir. Una operación de lectura lee la siguiente porción del archivo y automáticamente se avanza el puntero, llevando pista de la ubicación de I/O. Similarmente, una escritura añade en el final del archivo y avanza hasta el final del nuevo material añadido (el nuevo final del archivo). Con tal acceso, un archivo puede ser restaurado al inicio y, en algunos sistemas, un programa puede ser capaz de avanzar o retroceder *n* registros, para algún entero *n* (quizá para n = 1) (Figura 10.3). El acceso secuencial se basa en el modelo de cinta de un archivo, y trabaja bien en dispositivos de acceso secuencial como en los de acceso aleatorio.

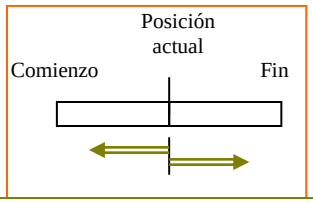


Figura 10.3 Archivo de acceso secuencial.

Acceso directo: otro método es el acceso direct. Un archivo esta constituido de registros lógicos de largo fijo que permite a los programas leer o escribir registros rápidamente sin un orden particular. El método de acceso directo esta basado en el modelo de disco de un archivo, ya que los discos permiten acceso aleatorio a cualquier bloque del disco. Para el acceso directo, los archivos se ven como una secuencia de bloques o registros numerados. Una archivo de acceso directo permite leer o escribir bloques arbitrarios. Así, podemos leer el bloque 14, luego el 53, y luego el 7. No existe restricción en el orden de las lecturas y las escrituras para los archivos de acceso directo.
Los archivos de acceso directo son de gran uso en grandes archivos donde la información se debe tener en un tiempo rápido. Las base de datos usan a menudo este tipo. Al llegar un pedido, se calcula que bloque contiene la respuesta, y luego se lee el bloque directamente para proveer la información deseada.
Se deben modificar las operaciones sobre archivos para incluir el número de bloque como parámetro. Ante esto tendremos, *read n* y *write n* (donde *n* es el número de bloque) en lugar de *read next* y *write next*. Una alternativa es la de guardar las operaciones del acceso secuencial *read next* y *write next*, y agregar la operación *position file to n*, donde *n* es el número de bloque. Así, para efectuar *read n* podríamos hacer *position file to n* y luego *read next*.
El número de bloque proveído por el usuario al sistema operativo es un número de bloque relativo, el cual es un índice referente al inicio del archivo. Así, el primer bloque relativo del archivo es el 0, el siguiente el 1, etc, aun aunque la actual dirección absoluta del bloque en el disco sea 14703 para el primer bloque y 3192

para el segundo. El uso de números de bloques relativos permite al sistema operativo decidir donde será ubicado el archivo. Algunos sistemas comienzan su número de bloque relativo en 0, otros en 1.
Dando un registro lógico de tamaño L, un pedido por el registro N será transformado en un pedido de I/O al lugar L * (N – 1) en el archivo. Ya que los registros lógicos son de tamaño fijo, es fácil leer, escribir o eliminar un registro.
Note que es fácil simular el acceso secuencial en un archivo de acceso directo. Solo se debe mantener una variable *cp*, el cual define nuestra posición actual, con lo que podemos implementar las operaciones para simular acceso secuencial en un archivo de acceso directo (Figura 10.4). Por otro lado, es extremadamente ineficiente y torpe simular un archivo de acceso directo en un archivo que es de acceso secuencial.

Acceso secuencial	Implementación para acceso directo.
reset	cp := 0;
read next	read cp; cp := cp + 1;
write next	write cp; cp := cp + 1;

Figura 10.4 Simulación de un acceso secuencial en un archivo de acceso directo.

Otros métodos de acceso: Los métodos adicionales generalmente involucran la construcción de un índice para el archivo. El índice contiene punteros a los diferentes bloques. Para encontrar una entrada en el archivo, primero se busca en el índice, y luego se usa el puntero para acceder al archivo directamente y encontrar la entrada deseada.
Por ejemplo, un archivo de precios por mayor puede listar el código universal de los productos (UPC) por ítem, con el precio asociado. Cada entrada consiste de un UPC de 10 dígitos y un precio de 6 dígitos, por lo que cada entrada es de 16 bytes. Si nuestro disco es de 1024 bytes por bloque, podemos almacenar 64 entradas por bloque (1024/16 = 64 entradas por bloque). Un archivo de 120.000 entradas ocupara aproximadamente 2000 bloques (120.000/64 = 1875 bloques = 1.920.000 bytes ≈ 2 millones de bytes). Si se mantiene el archivo almacenado por UPC, podemos definir un catálogo (tabla) consistente del primer UPC de cada bloque. El catálogo tendría 2000 entradas de 10 dígitos cada una (es una entrada para cada bloque, y cada entrada tiene el primer UPC de cada bloque), o 20.000 bytes, y así, podría ser mantenida en memoria. Para encontrar el precio de un ítem en particular, se debe hacer una búsqueda binaria en el catálogo. De ésta búsqueda, sabremos exactamente en que bloque se encuentra la entrada deseada y acceder a ese bloque. Así, este tipo de estructura nos permite hacer una búsqueda en archivos muy grandes haciendo poca I/O.
Con grandes archivos, el índice se podría trasformar demasiado grande para que sea mantenido en memoria. Una solución es crear un índice para el archivo de índices. El archivo de índices primario tendría punteros al archivo de índices secundario, que tendría punteros a los datos.
Por ejemplo, el método de acceso secuencial de catálogos de IBM (ISAM) usa un pequeño catálogo maestro que apunta a bloques de un catálogo secundario. Los bloques del catálogo secundario apuntan al actual archivo de bloques. El archivo esta almacenado ordenado por medio de una clave. Para encontrar un índice en particular, primero se hace una búsqueda binaria en el catálogo maestro, el cual provee el número de bloque del catálogo secundario. Este bloque es leído, y una vez más se hace una búsqueda binaria para encontrar el bloque conteniendo el registro deseado. Finalmente, se debe realizar una búsqueda secuencial de este bloque. De esta forma, cualquier registro puede ser ubicado por medio de su clave con, a lo sumo, dos lecturas de acceso directo.

Estructura del directorio

El sistema de archivo de las computadoras puede ser extenso. Algunos sistemas almacenan miles de archivos sobre cientos de gigabytes de disco. Para administrar estos datos se los necesita tener organizados. Esta organización se hace en dos partes. Primero, el sistema de archivos se divide en

particiones. Típicamente, cada disco contiene al menos una partición, el cual es una estructura de bajo nivel en el cual residen los archivos y los directorios.
Segundo, cada partición contiene información sobre los archivos que contiene. Esta información es mantenida en entradas en un dispositivo de directorio o directorio. El directorio almacena información (tal como nombre, lugar, tamaño y tipo) para todos los archivos en la partición. En la figura 10.6 se ve la organización típica de un sistema de archivo.

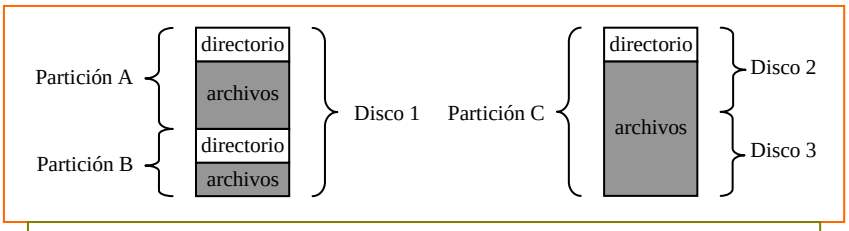


Figura 10.6 Una típica organización de un sistema de archivos.

El directorio se puede ver como una tabla de símbolos que traduce nombres de archivos en su respectiva entrada del directorio. El directorio se puede organizar de diferentes maneras. Se debe ser capaz de insertar una entrada, eliminar una entrada, buscar una entrada, y listar todas las entradas del directorio. En esta parte se examinaran varios esquemas para definir la estructura lógica del sistema de directorio. Cuando se quiere definir una estructura de directorio particular se deben tener en cuenta las operaciones que se le aplicaran al directorio:

- *Buscar un archivo:* Se debe escoger una estructura de directorio que sea capaz de encontrar la entrada en el directorio de un archivo particular. Ya que los archivos tienen nombres simbólicos y similares nombres pueden indicar relaciones entre archivos, se debe ser capaz de encontrar todos los archivos cuyo nombre coincida con un string en particular.
- *Crear un archivo:* se debe ser capaz de crear y agregar nuevos archivos en el directorio.
- *Eliminar un archivo:* cuando un archivo ya no se necesita más, se debe eliminar del directorio.
- *Listar un directorio:* se debe ser capaz de listar todos los archivos en un directorio, y los contenidos de la entrada del directorio para un archivo dado.
- *Renombrar un archivo:* ya que el nombre del archivo representa para el usuario el contenido del archivo, se debe ser capaz de cambiarles el nombre en caso de que su contenido varíe o su uso varíe. Esta función también debe permitir cambiar un archivo de posición en el sistema de directorio.
- *Recorrer el sistema de archivo:* se debe tener la posibilidad de recorrer y acceder a cada directorio, y a cada archivo en la estructura de directorio. Por confiabilidad, es buena idea almacenar los contenidos y la estructura completa del sistema de archivos a intervalos regulares. Esto consiste de copiar todos los archivos en cintas magnéticas. Esta técnica provee una copia de seguridad en caso de que el sistema falle o de que el archivo no se use más. En este caso el archivo puede ser copiado en la cinta y liberar el lugar que ocupaba en el disco para que su lugar sea reusado.

Directorio de único nivel: La estructura más simple de directorio es la de un solo nivel. Todos los archivos se encuentran en el mismo directorio (Figura 10.7). Este tipo de estructura tiene significantes limitaciones cuando el número de archivos es grande o cuando existe más que un usuario. Ya que todos los archivos están en el mismo directorio, los nombres de todos los archivos deben ser distintos.

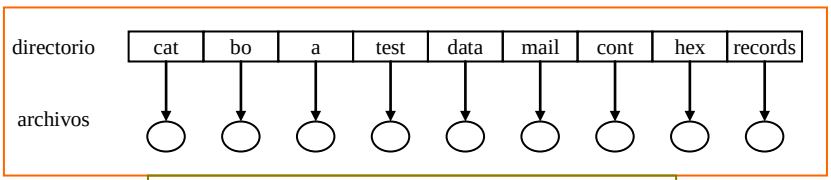


Figura 10.7 Directorio de único nivel.

Directorio de dos niveles: la mayor desventaja de una estructura de único nivel es que todos los archivos deben tener diferentes nombres para los diferentes usuarios. La solución es crear un directorio para cada usuario. En una estructura de dos niveles cada usuario tiene su directorio (UFD: user file directory). El UFD tiene una estructura similar pero lista solo los archivos de un único usuario. Cuando un usuario comienza a trabajar se busca en el directorio maestro del sistema (MFD: Master File Directory) su UFD, es decir, cada entrada en el MFD apunta a un UFD de algún usuario, y cada entrada del UFD apunta a los archivos (Figura 10.8).

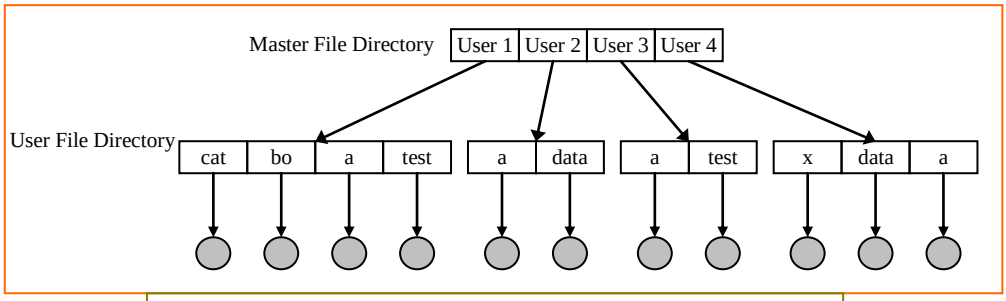


Figura 10.8 Estructura de directorio de dos niveles.

Cuando un usuario se refiere a un archivo en particular, solo se busca en su UFD. Así, diferentes usuarios pueden tener archivos con el mismo nombre, pero los archivos en un UFD deben tener nombres únicos. Para crear un archivo para un usuario, el sistema operativo busca en el UFD del usuario para ver si ya existe un archivo con ese nombre. De la misma manera, al eliminar un archivo el sistema operativo se limita a buscar sólo en el UFD del usuario. De esta manera no se eliminara accidentalmente un archivo de otro usuario.
Este tipo de estructura si bien resuelve el problema de archivos con nombres iguales para dos usuarios distintos, aun tiene problemas. Esta estructura aísla un usuario de otro, el cual es una ventaja cuando ,los usuarios son totalmente independientes, pero es una desventaja cuando usuarios quieren cooperar en una misma tarea y acceder a archivos entre sí.
Algunos sistemas no permiten que un usuario acceda a los archivos de otro. Si tales accesos son permitidos, un usuario debe poder nombrar un archivo del directorio de otro usuario. Para nombrar un archivo particular en una estructura de dos niveles únicamente se puede lograr dando tanto el nombre del usuario y el nombre del archivo. Así, el directorio de dos niveles se puede ver como un árbol. La raíz es el Master File Directory. Sus descendientes directos son los UFDs. Los descendientes de los UFDs son los archivos. Los archivos son las hojas del árbol. Especificando el nombre del usuario y el nombre del archivo se define un camino en el árbol desde la raíz (MFD) a la hoja (el archivo especificado). Cada archivo en el sistema tiene un camino único.
Un caso especial de esta situación ocurre con respecto a los archivos del sistema. Estos archivos son una parte del sistema (cargadores, asemblers, compiladores, librerías, etc.). Cuando se le da al sistema operativo el comando apropiado, estos archivos son leídos por el cargador y ejecutados. Muchos interpretes de comando actúan simplemente tratando al comando como el nombre de un archivo para cargar y ejecutar. Como se definió hasta ahora el sistema de directorio, este archivo podría ser buscado en el directorio del usuario actual. Una solución podría ser copiar los archivos del sistema en el directorio de cada usuario. Sin embargo, copiar todos los archivos del sistema podría provocar un gasto tremendo de lugar (si los archivos del sistema requieren 5 megabytes, entonces para soportar 12 usuarios requerirá 60 megabytes sólo para las copias de los archivos del sistema).
La solución estándar es la de usar un directorio de usuario especial que contiene los archivos del sistema (por ejemplo, user 0). Cuando se da el nombre de un archivo para ser cargado, el sistema operativo primero lo busca en el directorio del usuario actual. Si lo encuentra, lo usa. En caso contrario, el sistema automáticamente lo busca en el directorio de éste usuario especial que contiene los archivos del sistema. La secuencia de directorios buscados cuando se nombra un archivo es llamado camino de búsqueda (search path). Este método es le más usado en sistemas UNIX y MS-DOS.

Directorios estructurados como árboles: la organización natural es la de extender la estructura de directorio a un árbol de altura arbitraria (Figura 10.9). Esta generalización permite a los usuarios crear sus propios sub-directorios y organizar sus archivos acordemente. Por ejemplo, el sistema MS-DOS es estructurado como un árbol. De hecho, un árbol es la estructura más común para un directorio. Cada archivo en el sistema tiene un nombre de camino único. Un nombre del camino es desde la raíz, pasando por todos los subdirectorio hasta le archivo específico.

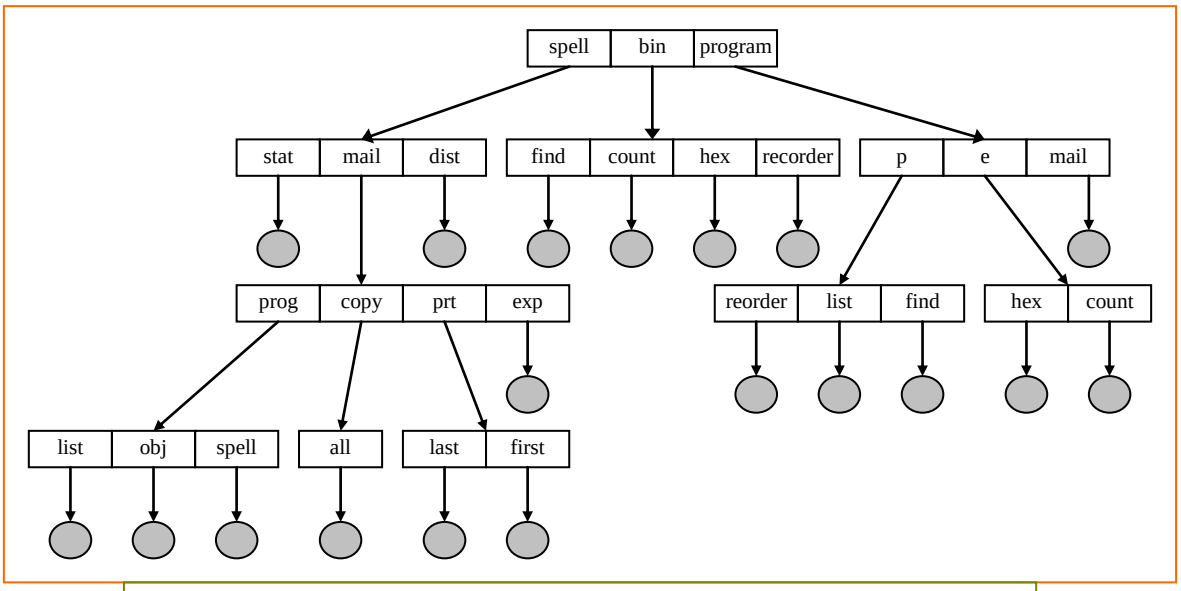


Figura 10.9 Estructura de un directorio estructurada como un árbol.

Un directorio (o subdirectorio) contiene un conjunto de archivos o subdirectorios. Un directorio es simplemente otro archivo, pero es tratado de una manera especial. Todos los directorios en el sistema tienen el mismo formato interno. Un bit en cada entrada del directorio define la entrada como un archivo (0) o como un subdirectorio (1). Llamadas al sistema especiales se usan para crear y eliminar directorios.
Cuando se hace una referencia a un archivo, se lo busca en el directorio actual. En caso de que se necesite un archivo que no esta en el directorio actual, entonces el usuario debe especificar el camino o cambiar el actual directorio y ubicarse en el directorio que esta el archivo. Al hacer este cambio de directorio actual, se provee una llamada al sistema (change directory) que toma como parámetro el nombre de un directorio y lo usa para redefinir el directorio actual.
Los nombres de los caminos pueden ser de dos tipos: caminos absolutos o caminos relativos. Un nombre de camino absoluto comienza de la raíz y continúa el camino hasta el nombre del archivo, dando los nombres de los directorios en el camino. Un nombre de camino relativo define un camino desde el directorio actual. Por ejemplo, en el directorio que se vio en la figura anterior, si el actual directorio es *root/spell/mail*, entonces el nombre de camino relativo *prt/first* se refiere al mismo archivo que el nombre de camino absoluto *root/spell/mail/prt/first*.
Una política de decisión importante es como manejar la eliminación de directorios. Si un directorio esta vació, entonces simplemente se puede eliminar su entrada en el directorio que lo contiene. Sin embargo, supongamos que el directorio a eliminar no esta vació (contiene varios archivos o subdirectorios). Entonces se pueden tomar dos caminos: Algunos sistemas tales como MS-DOS no eliminan el directorio a menos que este vació. Si hay varios subdirectorios, entonces el procedimiento de borrado debe ser hecho recursivamente sobre éstos. Un camino alternativo, tal como el que toma UNIX por medio del comando **rm**, provee la opción que, cuando se hace un pedido de eliminar un directorio, todos los archivos y subdirectorios que había en el directorio a eliminar también son eliminados. Esta política es más fácil de usar (ya que si tengo que eliminar muchos subdirectorios me los elimina todos de una vez), pero es peligrosa ya que puede que elimine archivos o subdirectorios que no quería eliminar.

En este tipo de estructuras de directorios, un usuario no solo puede acceder a sus archivos, sino también a los archivos de otro usuario, simplemente especificando el camino.
Note que el camino a un archivo en una estructura de directorio tipo árbol puede ser mucho más grande de dos niveles. Para permitir a los usuarios acceder a programas sin tener que recordar estos largos caminos, el sistema operativo Macintosh automatiza la búsqueda de archivos ejecutables. Lo que hace es mantener un archivo llamado "Desktop File", conteniendo el nombre y la ubicación de todos los programas ejecutables. Cuando se agrega un nuevo disco duro o disco blando al sistema, o se accede a la red, el sistema operativo recorre la estructura de directorio buscando por programas ejecutables en el dispositivo y graba dicha información. Este mecanismo soporta la ejecución por medio de hacer un doble clic en un archivo, causando que se lea el atributo de su creador, y se busque en el "Desktop File" para una coincidencia. Una vez que se produce la coincidencia, el programa ejecutable apropiado se inicia teniendo como entrada el archivo al que se le hizo el doble clic.

Directorios de grafos acíclicos: consideremos dos programadores que están trabajando en un proyecto. Los archivos asociados con este proyecto pueden ser almacenados en un subdirectorio, separándolos de los archivos y proyectos de los dos programadores. Pero ya que ambos programadores son igualmente responsables por el proyecto, ambos querrán que este directorio que contiene el proyecto este en su directorio. Entonces el subdirectorio en común podría ser compartido. Un archivo o directorio compartido existirá en el sistema de archivos en dos (o más) lugares a la vez. Note que un archivo (o directorio) compartido no es lo mismo que existan dos copias del mismo archivo en el sistema. Por medio de compartir archivos, solo existe una sola copia del archivo en el sistema, por lo que los cambios que una persona haga sobre el mismo aparecerán automáticamente en la vista que la otra persona tiene del archivo. Esta forma de compartir es muy importante en compartir subdirectorios: si una persona agrega un archivo en un subdirectorio compartido, el mismo aparecerá automáticamente en la vista que el otro (u otros) usuario tiene del subdirectorio.
Una estructura de árbol prohíbe compartir archivos o directorios. Un grafo acíclico permite que los directorios compartan archivos o subdirectorios (Figura 10.10).

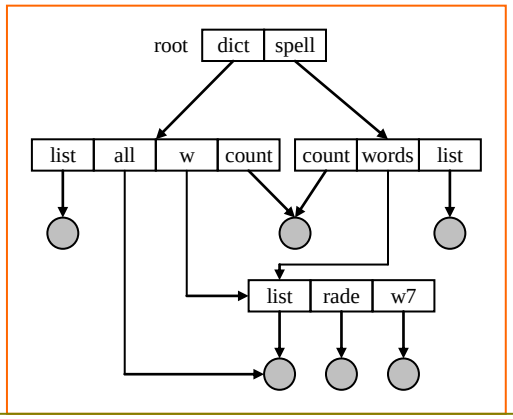


Figura 10.10 Estructura de directorio de grafo acíclico.

En una situación donde varias personas están trabajando como un equipo, todos los archivos que son compartidos pueden ser ubicados todos juntos en un directorio. Aun si existe solo un usuario, puede existir la situación donde se quiere que un mismo archivo aparezca en varios subdirectorios. Por ejemplo, un programa escrito para un proyecto en particular podría estar tanto en el directorio de todos los programas y en el directorio de los proyectos.
La forma de compartir directorios o proyectos se puede implementar de diferentes maneras. Una forma común, usada por muchos sistemas UNIX, es crear una nueva entrada de directorio llamada un link, el cual es un puntero a otro archivo o subdirectorio. Por ejemplo, un enlace puede ser implementado como un nombre de camino relativo u absoluto. Cuando se hace una referencia a un archivo, se busca en el directorio. Si la entrada en el directorio para este archivo esta marcada como un enlace (link), se devuelve

el nombre del archivo (o directorio) real. El enlace se implementa por medio de la copia del camino real (a los enlaces también se los denomina punteros indirectos).
La otra forma de implementar la compartición de archivos es simplemente duplicar toda la información en los subdirectorios compartidos. Así, ambas entradas son idénticas. En la forma de enlace, el enlace es claramente diferente de la entrada original (el enlace apunta a la entrada original), por lo que la entrada del enlace y la original no son iguales. En la forma de duplicar la información la entrada de la copia y del original son indistinguibles. El mayor problema con entradas de directorio duplicadas es mantener la consistencia en caso de que el archivo sea modificado.
Un problema con este tipo de estructuras involucra el recorrido. Ya que un archivo puede tener múltiples nombres de caminos absolutos, distintos nombres de archivos se pueden referir al mismo archivo. Si se trata de recorrer el sistema de archivos completo (para encontrar un archivo, o por motivos estadísticos, o para copiar todos los archivos en un backup), puede que recorramos estructuras de directorio compartidas más de una vez.
Otro problema con este tipo de estructuras involucra a la eliminación de archivos (o directorios), ya que en caso de que eliminemos un archivo que es compartido, puede que queden punteros apuntando a nada, y si además el espacio liberado es ocupado por otro archivo, ahora el puntero estará apuntando a un lugar incorrecto.
En un sistema donde se maneja la compartición de archivos por medio de enlaces simbólicos, esta situación es fácil de manejar. La eliminación de un enlace no necesita afectar el archivo original, solo se elimina el enlace. Si la entrada del archivo es la que se elimina, se libera el espacio de que ocupaba el archivo, pero los enlaces que apuntaban a este archivo se dejan pendientes. Se podría hacer una búsqueda de estos enlaces y eliminarlos, pero, a menos que se tenga una lista de enlaces asociados al archivo, ésta búsqueda puede ser muy costosa. Otra idea seria la de dejar el enlace pendiente hasta que se produzca un intento de usar el enlace. En ese momento, se determinara que el archivo asociado con dicho enlace no existe, y se tratara el acceso como cualquier intento de acceso ilegal. En este punto, se debe tener cuidado si el lugar fue reemplazo por otro archivo antes de que se haga referencia al enlace simbólico. En el caso de UNIX, se deja los enlaces cuando se elimina el archivo hasta que el usuario vea que el archivo ya no existe o que ha sido reemplazado.
Otra forma de eliminar es la de preservar el archivo hasta que todas las referencias al mismo sean eliminadas. Para implementarla, se debe tener algún mecanismo para determinar cuando se ha eliminado la ultima referencia al archivo. Se podría mantener una lista de todas las referencias al archivo (ya sean entradas de directorio o enlaces simbólicos). Cuando se establece un enlace o una copia de la entrada del directorio, una nueva entrada se agrega a la lista de referencias del archivo. Cuando se elimina en enlace o una entrada de directorio, se elimina ésta entrada de la lista. El archivo se elimina cuando su lista de referencias esta vacía. El problema con esta implementación es el tamaño que puede llegar a tomar esta lista. Sin embargo, no se necesita tener una lista completa, sino que solo se debe mantener un número con la cantidad de referencias que tiene el archivo. Así, cuando la cantidad llegue a cero se elimina el archivo.
El sistema operativo UNIX usa esta implementación para enlaces no simbólicos, manteniendo un contador de referencia en el bloque de información del archivo (o inode). Para evitar que el grafo pierda la propiedad de acíclico lo que se hace es prohibir que un directorio tenga más que una referencia, es decir, solo puede tener la referencia de su padre y nada más.
Para evitar el problema relacionado con la eliminación de archivos compartidos y los relacionados con la manutención de aciclicidad del grafo, el sistema operativo MS-DOS tiene una estructura de directorio de árbol, en vez de un grafo acíclico.

Directorio de grafo general: el problema serio en el uso de una estructura de grafo es el de mantener la propiedad de acíclico. En una estructura de árbol no existe problema de ciclos, pero cuando el sistema soporta enlaces, la estructura pasa a ser ya no la de un árbol sino la de un grafo.
La principal ventaja de un grafo acíclico es la simplicidad de los algoritmos para recorrer el grafo y para determinar cuando no existen más referencias a un archivo. Lo que se desea es evitar recorrer secciones compartidas de un grafo acíclico dos veces, principalmente por razones de performance. Si ya hemos

buscado un archivo particular en un gran directorio compartido sin haberlo encontrado, queremos evitar buscar en dicho directorio una vez más, ya que esta segunda búsqueda será solo un gasto de tiempo.
En caso de que se permitan ciclos en el grafo de directorio, un pobre algoritmo de búsqueda podría terminar en un loop infinito. Un problema similar existe cuando tratamos de determinar cuando se puede eliminar un archivo. Como en la estructura de directorio de grafo acíclico, podríamos tener un contador y en el momento que el contador llega a cero el archivo se puede eliminar. Pero puede existir el caso, si se permiten ciclos en el grafo, de que el contador no este en cero pero no se pueda hacer más referencia al archivo. Esta anomalía resulta por la posibilidad de referencia a uno mismo en la estructura de directorio. En éste caso, es necesario usar un esquema de garbage collection, el cual lo que hace es recorrer el sistema de archivos completo, marcando aquellos lugares que pueden ser accedidos. Luego, en un segundo paso junta todos los lugares que no fueron marcados y los ubica en una lista de espacio libre. Algoritmos similares son usados para asegurarse que en el momento de realizar un recorrido completo del sistema, solo se pase una sola vez por cada lugar. Lo que tiene este esquema es que es extremadamente costoso en cuanto al tiempo que consume, por lo que rara vez es invocado.
El garbage collection solo aparece ya que se permiten ciclos en el grafo, por lo que es mucho más fácil trabajar sobre grafos sin ciclos. En caso de que no se acepten ciclos, al intentar agregar un nuevo enlace, se deben invocar algoritmos que detectan ciclos en grafos. Igualmente, el costo es alto, especialmente cuando el grafo esta en disco. Generalmente las estructuras de árbol son más comunes que las de grafos acíclicos.

Protección

Cuando la información es mantenida en una computadora, la inquietud más importante es la protección de, tanto daños físicos (fiabilidad) como también accesos incorrectos (protección).
La fiabilidad se provee por medio de la duplicación de archivos. Muchas computadoras tienen programas de sistemas que automáticamente (o con la intervención del operador) copian los archivos en disco a cintas a intervalos regulares (una vez por día, o por semana, o por mes) para mantener una copia de seguridad en caso de que se produzca un destrozo accidental. Los sistemas de archivos pueden ser dañados por problemas de hardware (tal como errores en lecturas o escrituras), sobre tensión o falta de energía, caída del cabezal, suciedad, etc. Los archivos pueden ser eliminados accidentalmente, problemas en el software del sistema de archivo puede causar que se pierdan contenidos de los archivos.

Tipos de accesos: un sistema que no permite que un usuario acceda a los archivos de otro, no se necesita protección.
Los mecanismos de protección proveen el control de los accesos limitando los tipos de accesos que se pueden hacer a los archivos. Los accesos están permitidos o negados dependiendo de varios factores. Se pueden controlar varios tipos de operaciones diferentes:

- *Lectura:* leer desde un archivo.
- *Escritura:* escribir o rescribir un archivo.
- *Ejecutar:* cargar el archivo en la memoria y ejecutarlo.
- *Añadir:* escribir nueva información en el final del archivo.
- *Eliminar:* eliminar el archivo y liberar su espacio para un posible reuso.
- *Listar:* listar el nombre y atributos de un archivo.

Otras operaciones se pueden controlar, tales como renombrar, copiar, o editar el archivo.

Listar los accesos y agruparlos: la forma más común para resolver el problema de la protección es la de hacer los accesos dependiendo de la identificación del usuario. Varios usuarios pueden necesitar diferentes tipos de accesos a un archivo o directorio. El esquema más general para implementar los accesos dependiendo de su identificación es asociar a cada archivo y directorio una lista de accesos, especificando el nombre del usuario y los tipos de acceso permitidos para cada usuario. Cuando un usuario

pide el acceso a un archivo en particular, el sistema operativo chequea la lista de accesos asociada con el archivo. Si el usuario esta registrado para obtener el acceso pedido, entonces éste se realiza. De otra manera, ocurre una violación de protección, y se le niega el acceso al archivo.
El mayor problema con la lista de acceso es su largo. Si queremos que cualquier usuario pueda leer un archivo, entonces deberíamos tener una entrada para cada usuario en dicha lista para especificar que puede leer el archivo. Para reducir el largo de la lista de acceso, muchos sistemas reconocen tres clasificaciones de usuarios:

- *Owner:* el usuario que crea el archivo es el dueño.
- *Group:* un conjunto de usuarios quienes comparten el archivo y necesitan accesos similares es llamado grupo, o grupo de trabajo.
- *Universe:* los demás usuarios en el sistema constituyen el universo.

Como ejemplo, supongamos que una persona, Sara, esta escribiendo un libro. Ella ha contratado tres estudiantes para que le ayuden con el proyecto. El texto del libro es mantenido en un archivo llamado book. La protección asociada con tal archivo es:

- Sara debe ser capaz de poder realizar todas las operaciones sobre el archivo.
- Los estudiantes deben ser capaces solo de leer y escribir el archivo pero no de eliminarlo.
- Todos los demás usuarios solo pueden leer el archivo.

Para conseguir la protección, se debe crear un nuevo grupo, digamos text, con los tres estudiantes como miembros. El grupo text debe estar asociado con el archivo book, y se debe setear el tipo de acceso que tiene este grupo.
El número de miembros del grupo debe ser controlado firmemente para poder realizar un trabajo adecuado. Este control se puede conseguir de diferentes maneras. Por ejemplo, en UNIX, los grupos son creados y modificados solo por el administrador de la facilidad (o por un supervisor). Así, dicho control es conseguido por la interacción humana. En el sistema VMS, para cada archivo existe una lista de control de accesos que lista todos los usuarios que pueden acceder al archivo. El dueño del archivo puede crear y modificar esta lista.
Con esta clasificación limitada de protección, solo se necesitan tres campos para definir la protección. Cada campo es una colección de bits, donde cada uno permite o evita el tipo de acceso asociado a él. Por ejemplo, el sistema UNIX define tres campos de tres bits cada uno: **rwx**, donde **r** controla los accesos de lectura, **w** los de escritura, y **x** los de ejecución. Un campo es para el dueño del archivo, otro para los dueños del grupo, y el tercero para los usuarios restantes. En este esquema, se necesitan 9 bits por archivo para registrar la información de protección. Así, para nuestro ejemplo, los campos de protección para el archivo book son los siguientes: para la dueña Sara, los tres bits están seteados; para el grupo text, los bits **r** y **w** están seteados pero no el bit **x**; y para el resto, solo el bit **r** esta seteado.

Otras formas de protección: existe otra forma para el problema de protección, el cual es asociar a cada archivo un password. Así, los accesos al archivo son controlados por medio de este password, así como se hace con una computadora.
Igualmente, este esquema tiene varias desventajas. Primero, si asociamos un password para cada archivo, puede que el número de password que un usuario deba recordar sea muy grande, haciendo que el sistema no sea practico. Si solo existe un password para todos los archivos, una vez que se sabe el password, se puede acceder a todos los archivos.
La protección de archivos esta también disponible en sistemas de único usuario, tales como MS-DOS y Macintosh. Estos sistemas operativos cuando fueron diseñados ignoraban los problemas de protección. Sin embargo, ya que éstos sistemas están siendo ubicados sobre redes, donde es necesaria la compartición de archivos y la comunicación, los mecanismos de protección están siendo agregados.
Note que, en una estructura de directorio multinivel, no solo se necesita la protección de archivos individuales, sino que se necesita protección de colecciones de archivos existentes en un subdirectorio, es decir, se necesita protección de directorios. Las operaciones sobre directorios que deben ser protegidas

son diferentes a las de archivos. Lo que queremos es controlar la creación y eliminación de archivos en el directorio. Además, puede que queramos controlar si un usuario puede determinar la existencia de un archivo en un directorio. Ante esto, listar el contenido que tiene un archivo debe ser una operación protegida. Por lo tanto, si el nombre de un camino se refiere a un archivo en un directorio, el usuario debe estar autorizado para a acceder tanto al directorio como al archivo.

Un ejemplo: UNIX: en el sistema UNIX, la protección del directorio se maneja igual que la protección de archivos, es decir, con cada subdirectorio se asocia tres campos (dueño, grupo y universo), cada uno consistente de 3 bits (rwx). Así, un usuario puede listar el contenido de un subdirectorio sólo si el bit r esta seteado en el campo apropiado. Similarmente, un usuario puede cambiar su directorio actual solo si el bit x asociado con el subdirectorio esta seteado en el campo apropiado.
En la figura 10.12 se ve un ejemplo de lista de directorio en el entono UNIX. En el primer campo se describe la protección del archivo o directorio. Una letra d como el primer carácter indica que es un subdirectorio. También muestra el número de enlaces al archivo, el nombre del dueño, el nombre del grupo, el tamaño del archivo en bytes, el día de creación, y finalmente el nombre del archivo (con la extensión opcional).

-rw-rw-r-	1	pbg	staff	31200	Sep 3	08:30	intro.ps
drwx------	5	pbg	staff	512	Jul 8	09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8	09:35	doc/
drwxrwx---	2	pbg	student	512	Aug 3	14:13	student-proj/

Figura 10.12 Ejemplo de un directorio en UNIX.

11. Implementación del File-System

Como se vio anteriormente, el sistema de archivo provee los mecanismos para almacenar y acceder tanto a los datos como a los programas. El sistema de archivos reside permanentemente en almacenamiento secundario. En este capítulo se verán los problemas relacionados con el almacenamiento de archivos y accesos al medio más común de almacenamiento secundario: el disco. Se verán formas para signar espacio en disco, para recuperar espacio libre, para llevar pista de la ubicación de los datos, etc.

Estructura del sistema de archivos

Los discos proveen el lugar para almacenar el sistema de archivos. Para mejorar la eficiencia de I/O, la transferencia entre la memoria y el disco es realizada en unidades de bloques. Cada bloque esta en uno o más sectores. Dependiendo del drive de disco, los sectores varían desde 32 bytes a 4096 bytes; usualmente son de 512 bytes. Los discos tienen dos características importantes que los hacen un medio conveniente para almacenar los archivos:

1. Se pueden rescribir; es posible leer un bloque del disco, modificarlo y volver a escribirlo en el mismo lugar del disco.
2. Podemos acceder directamente a un bloque dado de información en el disco. Por eso, es simple acceder a cualquier archivo tanto aleatoria como secuencialmente, y cambiar desde un archivo a otro solo requiere mover la cabeza de lectura-escritura y esperar para que rote el disco.

Organización del sistema de archivo: Para proveer un acceso eficiente y conveniente al disco, el sistema operativo impone un sistema de archivo para permitir que los datos sean almacenados, ubicados y recuperados fácilmente. Un sistema de archivo plantea dos problemas de diseño bastante diferentes. Uno de ellos es como debe el usuario ver el sistema de archivo. Esta tarea involucra la definición de un archivo y sus atributos, las operaciones que se pueden hacer sobre él, y la estructura del directorio para organizar los archivos. El segundo es que se deben crear algoritmos y estructuras de datos para mapear el sistema de archivos lógico en el dispositivo de almacenamiento secundario físico.
El sistema de archivos esta compuesto de varios niveles. La estructura mostrada en la figura 11.1 es un ejemplo de un diseño en capas. Cada nivel en el diseño usa las características de los niveles inferiores para crear nuevas características para que las usen los niveles superiores.

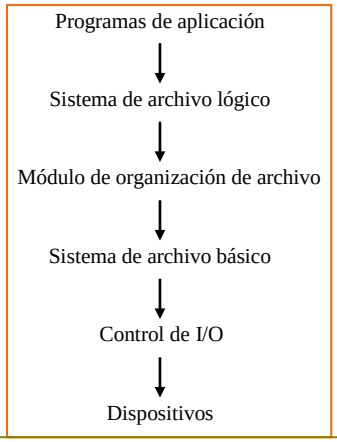


Figura 11.1 Sistema de archivos en capas.

El nivel más bajo, el control de I/O, consiste de drivers de dispositivos y manipuladores de interrupciones para transferir la información entre la memoria y el sistema de disco. Un driver de dispositivo puede ser pensado como un traductor. Su entrada consiste de comandos de alto nivel tal como "retrieve block 123" (recuperar bloque 123). Su salida consiste de instrucciones especificas de hardware de bajo nivel, las cuales son usadas por el controlador de hardware, el cual une el dispositivo de I/O con el resto del

sistema. El driver de dispositivo usualmente escribe patrones de bit específicos en una ubicación especial de la memoria del controlador de I/O para decirle al controlador en cual lugar del dispositivo actuar y que acciones tomar.
El sistema de archivo básico solo necesita emitir comandos genéricos para el driver del dispositivo apropiado para leer y escribir bloques físicos en el disco. Cada bloque físico esta identificado por su dirección numérica de disco (por ejemplo: drive 1, cilindro 73, pista 2, sector 10).
El módulo de organización de archivo conoce sobre los archivos y sus bloques lógicos, como así también de los bloques físicos. Conociendo el tipo de asignación de archivo usado y la ubicación del archivo, el módulo de organización de archivo puede traducir direcciones de bloques lógicas a direcciones de bloque físicas para que el sistema de archivo básico transfiera. Los bloques lógicos del archivo son numerados desde 0 (o 1) hasta N, mientras que los bloques físicos conteniendo el dato usualmente no son los mismos números que los bloques lógicos, por lo que la traducción es necesaria para ubicar cada bloque. El módulo de organización de archivo también incluye la administración del espacio libre, el cual lleva pista de los bloques libres y los provee a medida que le son pedidos.
Finalmente, el sistema de archivo lógico usa la estructura de directorio para proveer al módulo de organización de archivo con información que más tarde necesitara, dando un nombre simbólico de archivo. El sistema de archivo lógico también es responsable de la protección y seguridad.
Para crear un nuevo archivo, un programa de aplicación llama al sistema de archivo lógico. El sistema de archivo lógico conoce el formato de la estructura de directorio. Para crear un nuevo archivo, lee el directorio apropiado en memoria, lo cambia con la nueva entrada, y vuelve a escribir el directorio en el disco. Algunos sistemas operativos, incluyendo UNIX, tratan un directorio igual que un archivo, (hay un campo que indica que es un directorio). Otros sistemas operativos, incluyendo Windows NT, implementan llamadas al sistema diferentes para archivos y para directorios, y tratan a los directorios como entidades separadas de los archivos.
Ahora que el archivo se ha creado, puede ser usado para I/O. Para cada operación de I/O, la estructura de directorio puede ser explorada para encontrar el archivo, chequear los parámetros, ubicar sus bloques de datos, y finalmente realizar la operación sobre los bloques. Cada operación podría ocasionar un alto overhead. Más bien, antes de que el archivo pueda ser usado para procedimientos de I/O, éste debe ser abierto. Cuando es abierto, se recorre la estructura de directorio para encontrar la entrada de archivo deseada en el directorio. Parte de la estructura de directorio es almacenada en memoria para aumentar la velocidad de operaciones de directorio. Una vez que se encontró el archivo, la información asociada tal como el tamaño, el dueño, los permisos de acceso y la ubicación de los bloques de datos, es copiada en una tabla en memoria, la tabla de archivos abiertos, donde está toda la información de todos los archivos abiertos actualmente.
La primer referencia al archivo (normalmente un open) causa que la estructura de directorio sea recorrida y la entrada de directorio para el archivo sea copiada en la tabla de archivos abiertos. El índice de ésta tabla se retorna al programa del usuario, y todas las demás referencias se hacen a través de este índice en lugar de por medio del nombre. El nombre dado para el índice puede variar. Los sistemas UNIX se refieren a este como el *descriptor del archivo*, Windows NT como un *manejador de archivo*, y otros sistemas como un *bloque de control de archivo*. Consecuentemente, cuando el archivo es cerrado por todos los usuarios que lo han abierto, la información modificada del archivo es copiada en la estructura de directorio en disco.
Algunos sistemas utilizan múltiples niveles de tablas en memoria. Por ejemplo, en el sistema de archivo de UNIX BSD, cada proceso tiene una tabla de archivos abiertos que mantienen una lista de punteros. Cada entrada de esta lista apunta a una tabla de archivos abiertos del sistema (systemwide). Esta tabla contiene información fundamental de la entidad abierta. Para archivos, cada entrada de ésta tabla apunta a una tabla de inodes activos. Para otras entidades, tales como conexiones de red y dispositivos, apunta a información de acceso similar. La tabla de inodes activos es una tabla de inodes actualmente en uso, que esta en memoria, e incluye un campo que apunta a los bloques de datos en disco. En realidad, un open primero recorre la tabla de archivos abiertos para ver si el archivo ya esta abierto por algún otro usuario. Si lo esta, se crea una entrada en la tabla de archivos abiertos del proceso que apunta a la tabla de

archivos abiertos del sistema (systemwide). Si no lo esta, el inode es copiado en la tabla de inodes activos, se crea una nueva entrada en la systemwide para dicho archivo y se crea una nueva entrada en la tabla de archivos abiertos del proceso.

Carga del sistema de archivo: Así como un archivo debe ser abierto antes de ser usado, el sistema de archivos debe ser cargado antes de estar disponible para los procesos del sistema. El procedimiento de carga es el siguiente: se le da al sistema operativo el nombre del dispositivo, y la ubicación en la estructura de archivo al cual vincular el sistema de archivo (llamado el punto de carga, el cual es el lugar donde se cargara el sistema de archivo). Por ejemplo, en los sistemas UNIX, un sistema de archivo conteniendo los directorios del usuario (sea *home*), puede ser montado como */home*; luego, para acceder a la estructura de directorio en este sistema de archivo, uno podría preceder al nombre de directorio deseado con */home*, como por ejemplo, */home/jane*. Cargando este sistema de archivo bajo */users* resultara en el nombre de camino */users/jane* para alcanzar el mismo directorio.
Luego, el sistema operativo verifica que el dispositivo contiene un sistema de archivo valido. Lo hace analizando si el directorio tiene el formato esperado. Finalmente, el sistema operativo marca en su estructura de directorio que un sistema de archivo fue cargado en el punto de carga especificado. Este esquema permite al sistema operativo recorrer su estructura de directorio, intercambiando entre los sistemas de archivos a medida que sea apropiado.

Métodos de asignación

El acceso directo a los discos permite una gran flexibilidad en la implementación de archivos. En muchos casos, muchos archivos serán almacenados en el mismo disco. El mayor problema es como asignar espacio a estos archivos para utilizar eficientemente el espacio de disco y para poder acceder rápidamente a los archivos. Existen tres métodos principales de asignación de espacio de disco: contiguo, enlazado e indexado. Cada método tiene sus ventajas y desventajas. Algunos sistemas soportan los tres, pero es más común que todo el sistema soporte solo un método para todos los archivos.

Asignación contigua: El método de asignación contigua requiere que cada archivo ocupe un conjunto de bloques contiguos en el disco. Las direcciones de disco definen un orden lineal en el disco. Note que con este orden, acceder al bloque b+1 luego del b normalmente no requiere mover la cabeza. Cuando se necesita mover la cabeza (desde el último sector del cilindro al primer sector del otro cilindro, el cual es el caso donde el archivo no entro en un cilindro y la parte sobrante fue ubicada en el cilindro siguiente), el número de discos que se deben buscar para acceder al archivo ubicado contiguamente es mínimo.
La asignación contigua de un archivo esta definida por la dirección de disco y su largo (en unidades de bloques). Si el archivo es de n bloques de largo, y comienza en el lugar b, entonces el archivo ocupara los bloques b, b+1, ..., b+n-1. La entrada de directorio para cada archivo indica la dirección del primer bloque y el largo (Figura 11.3).

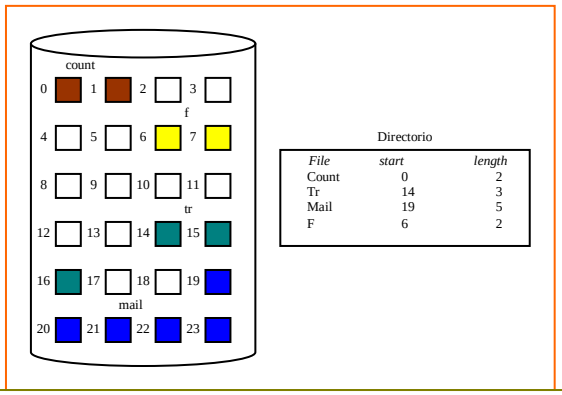


Figura 11.3 Asignación continua del espacio de disco.

Acceder a un archivo que fue ubicado contiguamente es fácil. Para acceso secuencial, el sistema de archivo recuerda la dirección de disco del último bloque referenciado y, cuando es necesario, lee el siguiente bloque. Para acceso directo al bloque i de un archivo que comienza en el bloque b, podemos acceder inmediatamente al bloque b+i. Así, tanto los accesos secuenciales como directos pueden ser soportados por la asignación contigua.
Una dificultad con la asignación contigua es encontrar espacio para un nuevo archivo. La implementación del sistema de administración del espacio libre de disco determina como se hace ésta tarea. Cualquier sistema de administración puede ser usado, pero algunos son más lentos que otros. Las estrategias más usadas para seleccionar un agujero vació de un conjunto de agujeros vacíos son *First-Fit* y *Best-Fit*, aunque ambos algoritmos sufre de fragmentación externa. Esto se produce ya que los archivos son creados y eliminados, provocando que vayan quedando espacios libres cuyo tamaño es insuficiente para cubrir un pedido. Antiguos sistemas aplicaban un procedimiento para eliminar la fragmentación. Lo que hacían era copiar el sistema de archivos completo en cinta o en discos blandos, se liberaba el espacio total del disco y luego se volvía a copiar la información desde el disquete al disco de manera contigua sin dejar espacios libres. La mayor desventaja era el costo de tiempo (el cual puede tomar horas).
Existe otro problema con la asignación contigua. El problema es determinar cuanto espacio necesitara un archivo. Cuando se crea el archivo, se debe buscar en el disco el espacio total necesitado, pero el problema es que es muy difícil saber cuanto espacio necesitara el archivo.
Si asignamos poco espacio al archivo, provocara que luego no pueda ser extendido. Especialmente con la estrategia Best-Fit, puede que ambos costados de donde se encuentra el archivo estén ocupados. Ante esto, existen dos posibilidades. La primera es que el programa del usuario sea terminado, con un apropiado mensaje de error. El usuario debe asignar más espacio y ejecutar el programa nuevamente. Como esta re-ejecución puede ser muy costosa, normalmente el usuario le da al archivo mucho más espacio del que piensa que necesitara provocando una gran perdida de espacio. La otra posibilidad es la de encontrar un nuevo espacio en donde ahora entre el archivo y liberar el espacio anterior. Con esta posibilidad, el usuario nunca se entera del problema de la memoria, aunque el programa se ejecutará más lentamente.
Aun si se conoce el espacio total que puede ocupar un archivo, la pre asignación puede ser ineficiente. Un archivo puede ir creciente de manera muy lenta hasta llegar a su tamaño final (si es que lo tiene, tardando quizá meses). Ante esto, si se le asigna a éste archivo el espacio total que necesita, una gran cantidad de espacio estará sin uso por un largo tiempo. A ésta cantidad de espacio que no se usa durante un largo tiempo es la fragmentación interna
Para evitar estos problemas algunos sistemas operativos usan un esquema de asignación continuo modificado, en el cual se asigna al archivo una cantidad de espacio suficiente. Cuando este espacio ya no es suficiente, se le asigna una extensión a la asignación anterior. Así, los bloques de un archivo están compuestos por un conjunto de bloques contiguos, más un enlace a la segunda cantidad asignada.

Asignación enlazada: la asignación enlazada resuelve todos los problemas de la asignación contigua. Con este tipo de asignación, cada archivo es una lista de bloques enlazados en disco, repartidos en cualquier lugar del disco. El directorio contiene un puntero al primer y último bloque del archivo (Figura 11.4). Cada bloque contiene un puntero al siguiente bloque. Estos punteros ocupan lugar en el bloque del usuario. Así, si cada bloque es de 512 bytes, y la dirección de disco requiere 4 bytes (el puntero), entonces los bloques del usuario serán de 508 bytes.

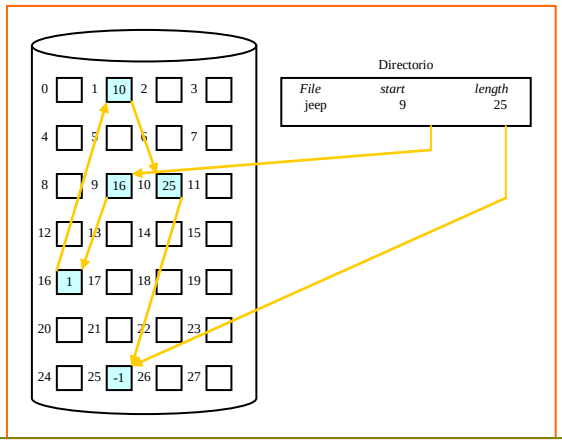


Figura 11.4 Asignación enlazada del espacio de disco.

Para crear un nuevo archivo, simplemente se crea una nueva entrada en el directorio. Con la asignación enlazada, cada entrada en el directorio tiene un puntero al primer bloque de disco del archivo. Este puntero esta inicialmente en nil para representar un archivo vació. El tamaño del archivo también se setea a 0. Una escritura en el archivo causa que sea buscado un bloque libre por medio de un sistema de administración de la memoria libre, este nuevo bloque es escrito, y es enlazado en el final del archivo. Para leer un archivo, simplemente se leen los bloques siguiendo los punteros de un bloque a otro.

No existe fragmentación externa con la asignación enlazada, y cualquier bloque libre en la lista de espacio libre puede ser usado para satisfacer un pedido. Note que tampoco existe necesidad de declarar el tamaño de un archivo cuando es creado. Un archivo puede continuar creciendo siempre y cuando existan bloques libres. Consecuentemente, tampoco es necesario realizar la compactación del disco.

Sin embargo, la asignación enlazada tiene desventajas. La mayor desventaja es que solo puede ser usada por los archivos de acceso secuencial. Para encontrar el i-ésimo bloque del archivo, se debe comenzar a recorrer desde el inicio del archivo, y seguir los punteros hasta que se llegue al bloque deseado. Cada acceso a un puntero requiere una lectura de disco, y a veces una búsqueda de disco. Consecuentemente, es ineficiente soportar acceso directo para la asignación enlazada de archivos.

Otra desventaja para la asignación enlazada es el espacio requerido para los punteros. Si un puntero requiere 4 bytes de los 512 del bloque, entonces el 0.78% del disco esta siendo usado para punteros, en lugar para información (512/4*100). Una solución usual para este problema es la de juntar varios bloques en unidades llamadas *clusters*, y asignar *clusters* en lugar de bloques. Por ejemplo, el sistema de archivos puede definir un cluster como 4 bloques, y operar en el disco solo con clusters. Así, los punteros usan un porcentaje mucho menor de espacio de disco. Su desventaja es que se incrementa la fragmentación interna, ya que si un cluster es parcialmente llenado, su espacio no usado es gastado. Los clusters pueden ser usados para mejorar el tiempo de acceso a disco de muchos algoritmos, por lo que este sistema es usado por la mayoría de los sistemas operativos.

Otro problema es el de la fiabilidad. Ya que los archivos están unidos por punteros esparcidos por todo el disco, en caso de que un puntero se pierda o sea dañado resultara un problema para el sistema operativo. Soluciones parciales son las de usar una lista doblemente enlazada o, almacenar el nombre del archivo y el número de block relativo en cada bloque; sin embargo, éstos esquemas requieren aun más overhead para cada archivo.

Una variación del método de asignación enlazada es la de usar una tabla de asignación de archivo (FAT: file-allocation table). Este simple pero eficiente método de asignación de espacio de disco es usado por los sistemas operativos MS-DOS y OS/2. Una sección de disco al inicio de cada partición tiene una tabla. Esta tabla tiene una entrada para cada bloque de disco, y esta organizada por número de bloque. La FAT es usada como una lista enlazada. La entrada de directorio contiene el número del primer bloque del archivo. La entrada de la tabla para éste número de bloque contiene entonces el número del siguiente bloque en el archivo. Esta cadena continúa hasta el último bloque, el cual tiene un valor especial de fin de archivo como la entrada de la tabla. Los bloques sin usar son indicados como un valor de tabla 0. Asignar un nuevo bloque

al archivo es simplemente encontrar el primer valor 0 como entrada en la tabla, y reemplazar el valor previo de fin de archivo con la dirección del nuevo bloque. El 0 es entonces reemplazado con el valor de fin de archivo (Figura 11.5).

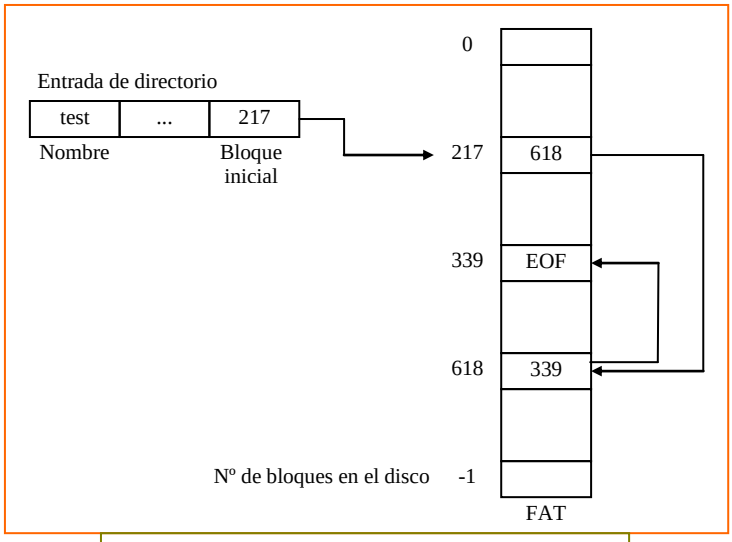


Figura 11.5 File-allocation table (FAT).

Note que el esquema de asignación FAT puede resultar en un significante número de búsquedas del cabezal del disco, a menos que la FAT este en la cache. La cabeza del disco se debe mover desde el comienzo de la partición para leer la FAT y encontrar la ubicación del bloque en cuestión, luego se mueve a la ubicación del bloque mismo. En el peor caso, ambos movimientos se harán para cada bloque. Un beneficio es que el tiempo de acceso aleatorio es mejorado, ya que la cabeza del disco puede encontrar la ubicación de cualquier bloque leyendo la información en la FAT.

Asignación indexada: la asignación enlazada resuelve los problemas de la fragmentación externa y los problemas de declaración de tamaño de la asignación contigua. Sin embargo, si no existe la FAT, la asignación enlazada no puede soportar el eficiente modo de accedo directo, ya que los punteros de los bloques están repartidos por todo el disco y necesitan ser recuperados en orden. La asignación indexada resuelve este problema poniendo todos los punteros juntos en una sola ubicación: el bloque de índices.
Cada archivo tiene se propio bloque de índices, el cual es un arreglo de direcciones de bloques de disco (Figura 11.6). La i-ésima entrada en el bloque de índices apunta al i-ésimo bloque del archivo. Este esquema es similar al esquema de paginación.

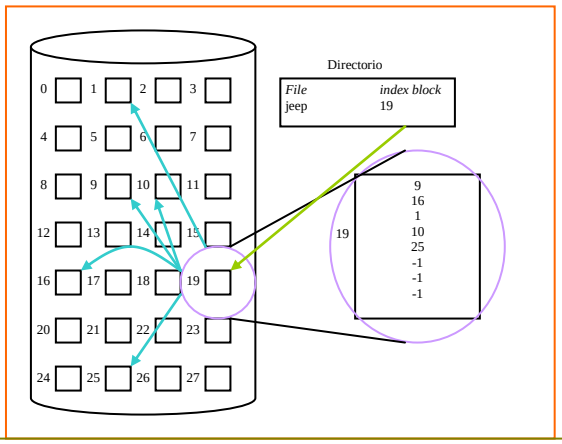


Figura 11.6 Asignación indexada del espacio de disco.

Cuando se crea el archivo, todos los punteros del bloque de índices están en nil. Cuando el i-ésimo bloque recibe el primer pedido, se obtiene un bloque libre del administrador de espacio libre, y su dirección es puesta en la i-ésima entrada del bloque de índices.
La asignación indexada soporta el acceso directo, sin sufrir fragmentación externa, ya que cualquier bloque libre puede satisfacer cualquier pedido.
La asignación indexada sufre de espacio gastado. El overhead de los punteros del bloque de índices es generalmente más grande que el overhead de los punteros de asignación enlazada. Consideremos un caso común en el cual tenemos un archivo con solo uno o dos bloques. Con la asignación enlazada, perdemos el espacio de solo un puntero por bloque (uno o dos punteros). Con la asignación indexada, un bloque de índices entero debe ser asignado, aun si uno o dos punteros no están en nil.
Ahora el problema es de que tamaño será el bloque de índices. Cada archivo debe tener un bloque de índices por lo que deseamos que el bloque de índices sea lo más pequeño posible. Sin embargo, si un bloque de índices es demasiado pequeño, no será capaz de almacenar los suficientes punteros que puede llegar a tener un gran archivo. Las posibilidades para tratar este tema son:

- *Esquema enlazado:* Un bloque de índices es normalmente un bloque de disco. Así, éste puede ser leído y escrito directamente. Para grandes archivos enlazamos varios bloques de índices juntos. Por ejemplo, un bloque de índice podría contener una pequeña cabeza que tiene el nombre del archivo, y un conjunto de las primeras cien direcciones de bloques de disco. La siguiente dirección (la última palabra en el bloque de índices) estará en nil (para un archivo pequeño), o es un puntero a otro bloque de índices (para grandes archivos).
- *Índice multinivel:* Una variante de la representación enlazada es la de usar un bloque de índices de primer nivel que apunta a un conjunto de bloques de índices de segundo nivel, los cuales apuntan a los bloques del archivo. Para acceder a un bloque, el sistema operativo usa el primer nivel de índices para encontrar el bloque de segundo nivel, y este bloque para encontrar el bloque de datos deseado. Esta alternativa podría ser continuada con un tercer o cuarto nivel, dependiendo del tamaño de archivo máximo deseado. Con bloques de 4096 bytes, se podría almacenar 1024 punteros de 4 bytes en un bloque de índices. Dos niveles de índices permiten 1.048.576 bloques de datos (1024 * 1024), el cual permite un archivo de hasta 4 gigabytes (Figura de abajo).
  Resolución: si tenemos 1.048.576 bloques y cada bloque es de 4Kb (4096 bytes), entonces tenemos 4.194.304 Kb de datos = 4096 Mb (4.194.304 Kb/ 1024 Kb) = 4 Gb de datos (4096 Mb / 1024 Mb).

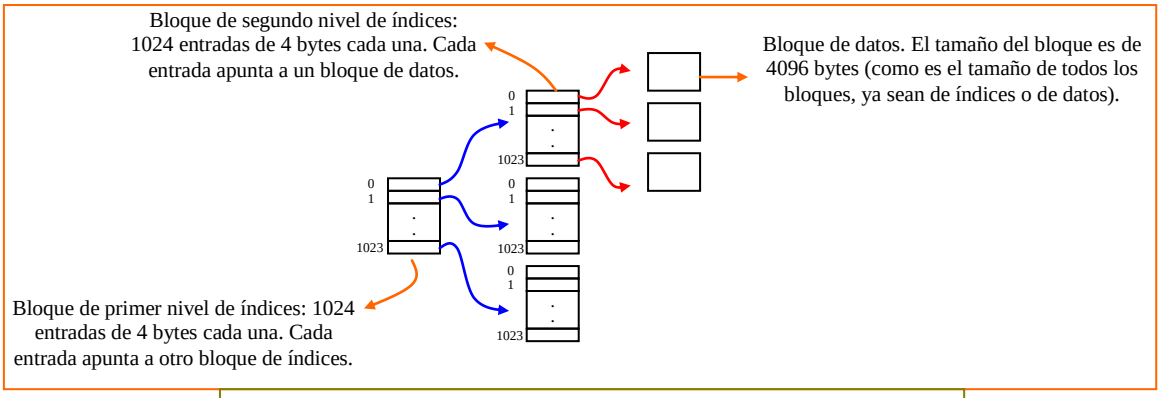


Figura que demuestra un indireccionamiento de dos niveles.

- *Esquema combinado:* otra alternativa, usada por el sistema BSD de UNIX, es mantener, digamos, los primeros 15 punteros a los bloques de índices en el bloque de índices del archivo (o inode). Los primeros 12 de estos punteros apuntan directamente a bloques, es decir, contienen direcciones de bloques que contienen datos del archivo. Así, los datos para pequeños archivos (no más de 12 bloques), no necesitan tener un bloque de índices separado. Si el tamaño del bloque es de 4K, entonces hasta 48K de datos del archivo pueden ser accedidos directamente. Los siguientes tres punteros apuntan a bloques indirectos. El primer puntero de bloque indirecto es la dirección de un

bloque de una sola indirección, es decir, es un bloque de índices conteniendo, no datos sino direcciones de bloques que contiene datos. Luego, el segundo de los tres es un bloque de punteros de dos indirecciones, el cual contiene la dirección de un bloque que contiene las direcciones de bloques que contiene punteros a los actuales bloques de datos. El último puntero de los tres contiene la dirección de un bloque de punteros de tres indirecciones. Bajo este método, el número de bloques que puede ser asignado a un archivo excede la cantidad de espacio direccionable por punteros de archivos de 4 bytes usado por muchos sistemas operativos. Un puntero de archivo de 32 bits alcanza solo 2^{32} bytes, o 4 gigabytes. Un inode se ve en la figura 11.7.

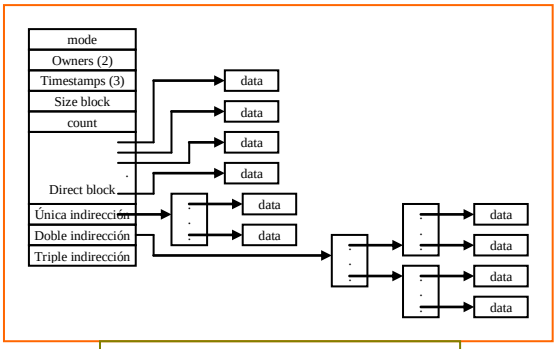


Figura 11.7 El inode UNIX.

Note que el esquema de asignación indexada sufre de algunos de los mismos problemas de performance que tiene la asignación enlazada. Especialmente, los bloques de índices pueden estar en memoria, pero los bloques de datos pueden estar dispersados por toda una partición.

Performance: los métodos de asignación que se vieron varían en la eficiencia de almacenamiento y en el tiempo que tardan para acceder a un bloque. Ambos son criterios importantes en el momento de seleccionar un método para el sistema operativo.
Una dificultad en el momento de comparar la performance de varios sistemas es determinar como los sistemas serán usados. Un sistema que hace la mayoría de los accesos secuencialmente podría usar un método diferente de aquel sistema que hace la mayoría de los accesos aleatoriamente. Para cualquier tipo de acceso, la asignación continua requiere solo un acceso para obtener el bloque de disco. Ya que podemos mantener fácilmente la dirección inicial del archivo en memoria, podemos calcular la dirección en el disco del i-ésimo bloque inmediatamente (o el siguiente bloque) y leerlo directamente.
Para la asignación enlazada, podemos también mantener la dirección del siguiente bloque en memoria y leerlo directamente. Este método es bueno para el acceso secuencial; sin embargo, para el acceso directo, un acceso al i-ésimo bloque requerirá i lecturas de disco. Este problema indica porque el método de asignación enlazada no es utilizado para una aplicación que requiere acceso directo.
Como resultado, algunos sistemas soportan acceso directo de archivos usando asignación contigua y acceso secuencial usando asignación enlazada. Para éstos sistemas, el tipo de acceso que se le aplicara al archivo debe ser declarado en el momento que se crea. Un archivo creado para accesos secuencial será enlazado y no podrá ser usado por el método de acceso directo. Un archivo creado para acceso directo será contiguo y soportara tanto el acceso directo como el secuencial, pero el largo total del archivo debe ser declarado cuando es creado. Note que, en este caso, el sistema operativo debe tener una estructura de datos y algoritmos apropiados para soportar ambos métodos de asignación. Los archivos pueden ser convertidos desde un tipo a otro por medio de la creación de un nuevo archivo del tipo deseado, en el cual se le copian los contenidos del viejo archivo. Así, el archivo viejo es eliminado, y el nuevo archivo es renombrado.
La asignación indexada es más compleja. Si el bloque de índices ya esta en memoria, entonces los accesos se pueden hacer directamente. Sin embargo, mantener el bloque de índices en memoria requiere considerable espacio. Si este espacio de memoria no esta disponible, primero se debe leer de disco el bloque de índices y luego el bloque de datos deseado. Para un indirecccionamiento de dos niveles, dos

lecturas a disco podrían ser necesarias. Así, la performance de la asignación indexada depende de la estructura del índice, del tamaño del archivo, y de la posición del bloque deseado.
Algunos sistemas combinan la asignación contigua con la asignación indexada, usando la asignación contigua para pequeños archivos (hasta tres o cuatro bloques), y automáticamente cambia a una asignación indexada si el archivo crece. Ya que la mayoría de los archivos son pequeños, la asignación contigua es eficiente para pequeños archivos.
Por ejemplo, la versión del sistema operativo de UNIX de Sun Microsystems fue cambiada en 1991 para mejorar la performance en los algoritmos de asignación del sistema de archivos.
Muchas optimizaciones están en uso. Por la disparidad entre la velocidad de la CPU y el disco, no es irrazonable agregar miles de instrucciones extras al sistema operativo para salvar solo unos pocos movimientos de la cabeza del disco. Además, esta disparidad se esta aumentando cada vez más, por lo que también es razonable agregar ciento de miles de instrucciones para salvar algún movimiento del cabezal.

Administración del espacio libre

Ya que existe una cantidad limitada de espacio en el disco, es necesario reusar el espacio de archivos eliminados para archivos nuevos, en caso de que sea posible (en los discos ópticos solo se permite una escritura en un sector dado, por lo que no se puede reusar por motivos físicos). Para mantener pista del espacio de disco libre, el sistema mantiene una lista de espacios libres, el cual registra todos los bloques de disco que están libres (aquellos que no están asignados a archivos o directorios). Para crear un archivo, se debe buscar en la lista de espacios libres por el espacio requerido, y se le asigna el espacio al nuevo archivo. Este espacio es entonces eliminado de la lista de espacios libres. Al eliminarse un archivo, su espacio de disco es agregado a la lista de espacios libres. La lista de espacios libres, a pesar de su nombre, puede no ser implementado como una lista.

Vector de bit: Frecuentemente, la lista de espacios libres es implementada como un *mapa de bits* o *vector de bits*. Cada bloque es representado por 1 bit. Si el bloque esta libre, el bit esta en 1; si el bloque esta asignado, el bit esta en 0.
La mayor ventaja de este método es que es relativamente simple y eficiente encontrar el primer bloque libre, o n bloque libres consecutivos en disco. De hecho, muchas computadoras proveen instrucciones de manejo de bits que pueden ser usadas para este propósito. Por ejemplo, la familia Intel comenzó con el 80386 y la familia Motorola comenzó con el 68020 (procesadores que han impulsado sistemas PCs y Macintosh, respectivamente) tienen instrucciones que retornan el offset en una palabra del primer bit con el valor 1. De hecho, el sistema operativo Macintosh Apple usa el método de vector de bit para asignar el espacio en disco. Para encontrar el primer bloque libre, el sistema operativo de Macintosh chequea secuencialmente cada palabra en el vector de bits para ver si el valor no es cero, ya que un valor 0 de palabra tiene todos sus bits en cero y representa un conjunto de bloques asignados. La primer palabra que no sea cero es analizada para ver donde esta el primer bit en 1, el cual es el lugar del primer bloque libre. Para calcular el número del bloque es:

(número de bits por palabra) * (número de palabras con valor 0) + offset del primer bit en 1.

Desafortunadamente, los vectores de bit son ineficientes a menos que el vector de bits completo este completamente en memoria principal. Mantenerlo en la memoria principal solo es posible para pequeños discos, tales como microcomputadoras, pero no para grandes. Un disco de 1.3 gigabytes con bloques de disco de 512 necesitara un vector de bits de más de 310K para llevar pista de los bloques libres. El clustering de bloques en grupos de cuatro reducirá este número a 78K por disco.

Lista enlazada: otro método es el de enlazar todos los bloques libres, manteniendo un puntero al primer bloque libre en una ubicación especial del disco y en memoria. El primer bloque contiene un puntero al siguiente bloque de disco libre, y así (Figura 11.8). Sin embargo, este esquema no es eficiente. Para

recorrer la lista, se debe leer cada bloque, el cual requiere substancial tiempo de I/O. Afortunadamente, recorrer la lista de espacios libres no es una acción habitual. Usualmente, el sistema operativo simplemente necesita un bloque de espacio libre, por lo que solo basta con fijarse donde se encuentra el primer bloque.

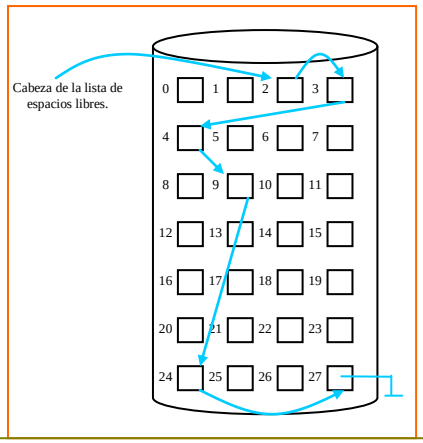


Figura 11.8 Lista enlazada de espacio libre sobre el disco.

Agrupación: una modificación del método de la lista de bloques libres es la de almacenar las direcciones de n bloques libres en el primer bloque libre. Los primeros n-1 de estos bloques están actualmente libres. El último bloque contiene las direcciones de otros n bloques libres y así. La importancia de esta implementación es que las direcciones de un gran número de bloques libres pueden ser encontrada rápidamente.

Contadores: otro método es tomar ventaja del hecho de que, generalmente, varios bloques contiguos pueden ser asignados o liberados simultáneamente, particularmente cuando los bloques son asignados con el mecanismo de asignación contigua o por medio de clusters. Así, en lugar de tener una lista de n direcciones de bloques libres de disco, se puede mantener la dirección del primer bloque libre de una secuencia y el tamaño de la secuencia. Cada entrada en la lista de espacios libres consiste entonces de una dirección y de una cantidad.

Implementación del directorio

La selección de algoritmos de asignación de directorio y administración de directorio tienen un efecto importante sobre la eficiencia, performance y fiabilidad del sistema de archivos. Por lo tanto, es importante entender los tradeoffs involucrados en éstos algoritmos.

Lista lineal: el método más simple de implementar un directorio es usar una lista lineal de nombre de archivos los cuales apuntan a los bloques de datos. Una lista lineal de entradas de directorio requiere una búsqueda lineal para encontrar una entrada en particular. Este método es simple de programar pero consume mucho tiempo al ejecutarlo. Para crear un nuevo archivo, primero se debe explorar en el directorio para estar seguros de que no existe un archivo con el mismo nombre. Luego, se agrega la entrada al final del directorio. Para eliminar un archivo, se busca en el directorio el archivo y se libera el espacio que tenia asignado. Para reusar la entrada del directorio se pueden tomar varios caminos. Se puede marcar la entrada como no usada (asignándole un nombre especial a cada entrada no usada), o se puede agregar dicha entrada a una lista de entradas libres de directorio. Una tercer alternativa es la de copiar la última entrada del directorio en la ubicación liberada, y decrementar el largo del directorio, disminuyendo así los tiempos de las operaciones.
La desventaja que tiene éste método de representar el directorio es el tiempo lineal de búsqueda que tiene. Muchos sistemas operativos implementan que la información de los directorios más recientemente usados esté en cache. Un éxito de cache evita constantes re lecturas a disco. Una lista ordenada permite

búsquedas binarias (árbol) decrementando el tiempo de búsqueda promedio. Sin embargo, se requiere que la lista este ordenada, provocando que en las eliminaciones y creaciones de archivos se deba mover grandes cantidades de información de directorio para mantener ordenada la lista. Una estructura de árbol más sofisticada, tal como árbol B, podría ayudar.

Tabla de hash: otra estructura que se está usando para el directorio es la tabla de hash. En este método, una lista lineal almacena las entradas del directorio, pero también se usa una estructura de dato hash. La tabla de hash toma un valor calculado a partir del nombre de archivo y retorna el puntero al nombre del archivo en la lista lineal. Por lo tanto, esto provocara que se decremente el tiempo de búsqueda. La inserción y eliminación son también simples aunque se deben tener en cuenta algunas cuestiones sobre colisiones (situaciones donde dos archivos tienen la misma ubicación en la tabla de hash). La mayor dificultad con la tabla de hash es que generalmente su tamaño es fijo y su dependencia de la función de la tabla de hash con el tamaño que tendrá ésta tabla.

Eficiencia y performance

Ahora que se ha discutido las opciones de asignación de bloques y administración del directorio, se verán sus efectos sobre la performance y eficiencia en el uso del disco. Los discos suelen ser el mayor problema en la performance del sistema ya que son lentos. En esta parte se verán algunas técnicas que mejoran la eficiencia y performance del almacenamiento secundario.

Eficiencia: el uso eficiente del espacio en disco es muy dependiente de los algoritmos de asignación del disco y directorio que se están usando. Por ejemplo, los inodes UNIX están preasignados en una partición. Aun un disco vació tiene un porcentaje de su espacio perdido para inodes. Sin embargo, preasignando los inodes y desparramarlos por la partición, se mejora la performance del sistema de archivo. Esta mejora es el resultado de los algoritmos de asignación y espacio libre de UNIX, el cual trata de mantener los bloques de datos cercanos al bloque inode del archivo para reducir el tiempo de búsqueda.
Como otro ejemplo, veamos nuevamente el esquema de clustering, el cual ayuda en la performance de la búsqueda del archivo y la transferencia del mismo, contra el costo de la fragmentación interna. Para reducir esta fragmentación, el BSD de UNIX varia el tamaño del cluster. Grandes clusters son usados cuando pueden ser llenados, y los pequeños clusters son usados para pequeños archivos y el último cluster de un archivo.
También se debe considerar los tipos de datos mantenidos en la entrada del directorio del archivo (inode). Comúnmente, el día de la última escritura se graba para proveer información al usuario y determinar si se le debe hacer al archivo una nueva copia de seguridad. Algunos sistemas también mantienen el día del último acceso. El resultado de mantener esta información es que cuando el archivo es abierto para lectura, se debe cambiar la entrada en el directorio para este archivo. Este cambio requiere que el bloque sea traído a memoria, cambiar la sección, y volver el bloque a disco (ya que las operaciones sobre discos ocurren solo en bloques o clusters). Esto puede provocar una ineficiencia cuando son muy frecuentes los accesos a archivos, por lo que se debe balancear su beneficio contra el costo de performance cuando se diseña el sistema de archivo.
Como ejemplo, consideremos como afecta a la eficiencia la elección del tamaño de los punteros usados para acceder a los datos. La mayoría de los sistemas usan o punteros de 16 bits o de 32 bits. Éstos tamaños de punteros limitan el largo del archivo a 2^{16} (64K) o 2^{32} bytes (4 Gb) respectivamente. Algunos sistemas implementan direcciones de 64 bits para incrementar este límite a 2^{64} bytes, el cual es un número muy grande. Sin embargo, los punteros de 64 bits toman más espacio para almacenarse, por lo que hacen que los métodos de administración y asignación del espacio libre (listas enlazadas, catálogos, etc.) usen más espacio de disco.
Como otro ejemplo, consideremos la evolución del sistema operativo Solaris de Sun. Originalmente, muchas estructuras de datos eran de tamaño fijo. Estas estructuras incluían la tabla de procesos y la tabla de archivos abiertos. Cuando la tabla de procesos estaba llena, no se podían crear más procesos. Cuando la

tabla de archivos se llenaba, no se podían abrir más archivos. El sistema fracasaba al proveer servicio a los usuarios. El tamaño de estas tablas solo se podía incrementar si se recompilaba el kernel y reiniciando el sistema. Desde la liberación de Solaris 2, casi todas las estructuras del kernel son de asignación dinámica. Por supuesto, los algoritmos que manipulan éstas tablas son más complicados, y el sistema operativo es un poco más lento, ya que debe asignar y desasignar dinámicamente las entradas de las tablas.

Performance: Una vez que se seleccionan los métodos básicos de disco, existen aun varias formas de mejorar la performance. Como se vio, la mayoría de los controladores de disco incluyen memoria local para formar una cache en el controlador, el cual es suficientemente grande para almacenar pistas enteras a la vez. Una vez que la búsqueda se hizo, la pista es ubicada en la cache del disco comenzando en el sector que está debajo de la cabeza (aliviando el tiempo de latencia). Luego el controlador del disco transfiere cualquier sector pedido al sistema operativo, es decir, se transfiere desde el controlador a la memoria principal. Algunos sistemas mantienen una sección separada de la memoria principal destinada para cache de disco, donde los bloques son mantenidos bajo la suposición de que serán usados en poco tiempo. LRU es un algoritmo razonable para el reemplazo de bloques. Otros sistemas, (tal como la versión UNIX de Sun), tratan a toda la memoria física (es decir a la memoria principal) no usada como una pileta de buffers que esta compartida por el sistema de paginado y el sistema de caching de los bloques de disco. Un sistema que realiza muchas operaciones de I/O usara la mayor parte de su memoria como una cache de bloques, mientras que un sistema que ejecuta muchos programas usara la mayor parte de la memoria como espacio de paginado.
Algunos sistemas optimizan sus cache de disco (en memoria principal, es decir, la parte de la memoria destinada para los bloques) usando diferentes algoritmos de reemplazo, dependiendo del tipo de acceso al archivo. Un archivo que esta siendo leído o escrito secuencialmente no deberá tener un algoritmo de reemplazo LRU, ya que el bloque más recientemente usado será el último. En cambio, el acceso secuencial puede ser optimizado por las técnicas conocidas como *free-behind* y *read-ahead*. Free-behind elimina un bloque desde el buffer tan pronto como se pide el siguiente bloque. El bloque anterior es probable que no se use nuevamente por lo que gasta espacio de buffer. Con read-ahead, un bloque pedido y una subsecuencia de bloques son leídos y puestos en dicha cache de disco, ya que es probable que ésta subsecuencia se pida luego de usar el primer bloque. Recuperar todos éstos bloques de una sola transferencia ahorra gran cantidad de tiempo.
Otro método usado para mejorar la performance de la memoria principal es común en computadoras personales. Una sección de memoria es seteada aparte y tratada como un disco virtual, o un disco RAM. En este caso, el driver del dispositivo del disco RAM acepta todas las operaciones estándar de disco, pero realiza todas estas operaciones en la sección de memoria, en lugar de en el disco. Ante esto, se pueden ejecutar todas las operaciones de memoria sobre este disco RAM y, excepto por la gran velocidad, los usuarios no notaran una diferencia. Desafortunadamente, los discos RAM son útiles para almacenamiento temporario, ya que una falla de energía o un reinicio del sistema las borrara. Comúnmente, los archivo temporarios tales como los archivos intermedios de compiladores se almacenan ahí.
La diferencia entre un disco RAM y una cache de disco es que los contenidos del disco RAM son totalmente controlados por el usuario, mientras que la cache de disco esta totalmente controlada por el sistema operativo (ambas están en la memoria principal). Por ejemplo, una disco RAM estará vacío hasta que el usuario (o programas) creen archivos allí. La figura 11.9 muestra una posible ubicación de cache en un sistema.

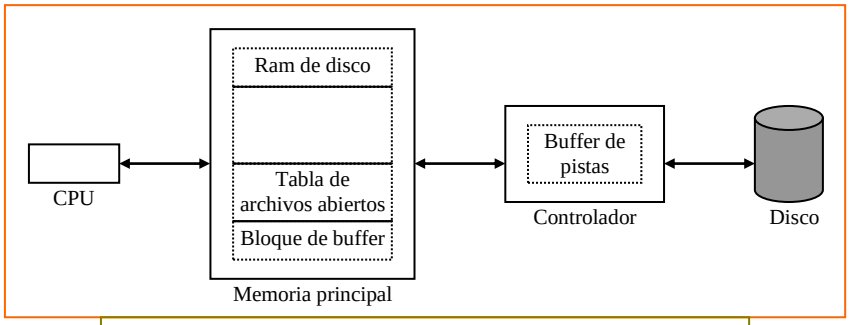


Figura 11.9 Varias ubicaciones de disk-caching.

Recuperación

Ya que tanto los archivos como los directorios son mantenidos en memoria principal y en disco, se debe tener seguridad de que fallos en el sistema no provocaran perdidas de datos o inconsistencia.

Chequeo de consistencia: Como se vio, parte de la información del directorio esta en memoria principal (cache) para aumentar la velocidad de acceso. La información del directorio en la memoria principal es generalmente información más actualizada de la que esta en disco, ya que las escrituras de la información del directorio en la cache al disco no necesariamente ocurre tan pronto como el cambio toma lugar.
Consideremos el posible efecto de una caída del sistema. En este caso, la tabla de archivos abiertos generalmente se pierde, y can ésta, todos los posibles cambios realizados a los directorios de archivos abiertos. Esto puede dejar al sistema de archivos en un estado inconsistente. El actual estado de los archivos no es el que esta descrito en la estructura de directorio. Frecuentemente, se corre un programa especial en el momento de reinicio para chequear alguna inconsistencia de disco.
El chequeo de consistencia compara el dato en la estructura de directorio con el dato en los bloques de disco, y trata de arreglar cualquier inconsistencia que encuentre. Los algoritmos de asignación y administración de memoria libre dictan que tipos de problemas se pueden encontrar y como se pueden arreglar con éxito éstos problemas. Por ejemplo, si se usa la asignación enlazada y hay un enlace desde un bloque al siguiente, entonces el archivo completo puede ser reconstruido a partir de los bloques de datos, y la estructura de directorio puede ser recreada. La perdida de un entrada de directorio completa en un sistema de asignación indexada podría ser desastrosa, ya que un bloque de dato no tiene conocimiento de otro. Por esta razón, UNIX almacena en cache (memoria principal) las entradas de directorio para lecturas, pero cualquier escritura de dato causara que el bloque inode sea escrito en disco antes de que aparezca en el correspondiente bloque de dato.

Backup y restauración: ya que los discos magnéticos a veces fallan, se debe tener mucho cuidado de no perder los datos para siempre. Con este fin, programas del sistema pueden ser usados para que realicen backups de datos desde el disco a otro dispositivo de almacenamiento, tal como discos blandos, discos magnéticos, o discos ópticos.
Para minimizar la necesidad de copiado, podemos usar la información de cada entrada de archivo del directorio. Por ejemplo, si el programa que realiza el back up conoce cuando se hizo la última copia, y el último día que fue modificado el archivo, puede darse cuenta de que el archivo no fue modificado, por lo que no necesita copiarlo nuevamente.

IV. Sistemas de I/O

Los dispositivos que se vinculan con una computadora son muy variados. Los dispositivos transfieren un carácter o un bloque de caracteres a la vez. Se los puede acceder solo secuencial o aleatoriamente. Transfieren datos sincrónica o asincrónicamente. Pueden ser compartidos o no. Pueden ser solo de lectura o de escritura-lectura. Varían en gran manera su velocidad. En muchas formas ellos son los componentes más lentos de la computadora.
Por la variación de todos éstos dispositivos, el sistema operativo necesita proveer una forma de funcionalidad para aplicaciones, que les permita controlar todos los aspectos de los dispositivos. Un objetivo clave del subsistema de I/O del sistema operativo es el de proveer una interfaz lo más simple posible al resto del sistema. Ya que los dispositivos son "el cuello de botella" de la performance, otro aspecto clave es optimizar la I/O para maximizar la concurrencia.

12. Sistemas de I/O

Los dos trabajos más importantes de una computadora son la I/O y el procesamiento. En muchos casos, el trabajo más importante es el de I/O, el cual incide en el procesamiento.
El rol del sistema operativo en computar la I/O es el de administrar y controlar las operaciones de I/O y los dispositivos de I/O. Primero, en este capítulo se describirá el hardware básico de I/O. Luego, se discutirán los servicios de I/O que brinda el sistema operativo, y la incorporación de estos servicios en la aplicación de interfase de I/O.

Vistazo

El control de los dispositivos conectados a la computadora es uno de los temas más importantes en el momento de diseñar el sistema operativo. Ya que los dispositivos de I/O varían en cuanto a su función y velocidad (considere un mouse, un disco duro, un CD-ROM), se necesita una gran variedad de métodos para controlarlos. Estos métodos forman el subsistema de I/O del kernel, el cual separa el resto del kernel de la complejidad de administrar los dispositivos de I/O.
La tecnología de los dispositivos de I/O muestra dos conflictos. Por un lado, se ve que se está estandarizando las interfaces de software y hardware. Esta tendencia ayuda a incorporar una generación de mejores dispositivos en computadoras y sistemas operativos ya existentes. Por otro lado, se ve un gran incremento de la variedad de dispositivos de I/O. Los elementos de hardware básicos, tales como puertos, buses, y controladores de dispositivos, proveen de una gran variedad de dispositivos de I/O. Para encapsular los detalles y las rarezas de cada dispositivo, el kernel del sistema operativo está estructurado para usar módulos device driver. Los drivers de dispositivos presentan una interfase uniforme de acceso al dispositivo al subsistema de I/O, así como las llamadas al sistema proveen una interface estándar entre la aplicación y el sistema operativo.
En este capítulo, se verán los mecanismos de hardware básicos que realizan la I/O, y la forma en que el sistema operativo organiza los dispositivos de I/O en categorías para formar una aplicación general de interfase de I/O. Se verán también los mecanismos del kernel que sirven de puente entre el hardware de I/O y software de aplicación.

Hardware de I/O

Las computadoras operan con una gran variedad de tipos de dispositivos. Los tipos generales incluyen dispositivos de almacenamiento (discos, cintas), dispositivos de transmisión (tarjetas de red, módems), y dispositivos de interfase humana (pantalla, teclado, mouse). Otros dispositivos son más especializados.
A pesar de la increíble variedad de dispositivos de I/O que pueden ser usados con una computadora, solo se necesitan pocos conceptos para entender como se vinculan, y como el software puede controlar el hardware.

Un dispositivo se comunica con el sistema de la computadora enviando señales sobre un cable o aun sobre el aire. El dispositivo se comunica con la maquina por medio de un punto de conexión llamado puerto (por ejemplo, un puerto serial). Si uno o más dispositivos usan un conjunto de cables en común, la conexión se llama bus. En términos más específicos, un bus es un conjunto de cables con un protocolo bien definido que especifica el conjunto de mensajes que pueden ser enviados por él. En términos electrónicos, el mensaje es un conjunto de voltajes eléctricos que se le aplican al cable. Cuando el dispositivo A tiene un cable que se conecta con el dispositivo B, y B tiene un cable que se conecta con el dispositivo C, y el dispositivo C tiene un puerto en la computadora, este arreglo es llamado *daisy chain*. Esto usualmente opera como un bus.
Los buses son muy usados en la arquitectura de la computadora. En la figura 12.1 se ve una estructura de bus típica de una PC. En ella se ve un bus PCI (el bus común del sistema), que conecta el subsistema de memoria del procesador a los dispositivos rápidos, y un bus de expansión que conecta dispositivos más lentos tal como el teclado y puertos seriales y paralelos. En la parte de arriba a la derecha de la figura, hay cuatro discos conectados juntos sobre un bus SCSI que esta conectado a un controlador SCSI.

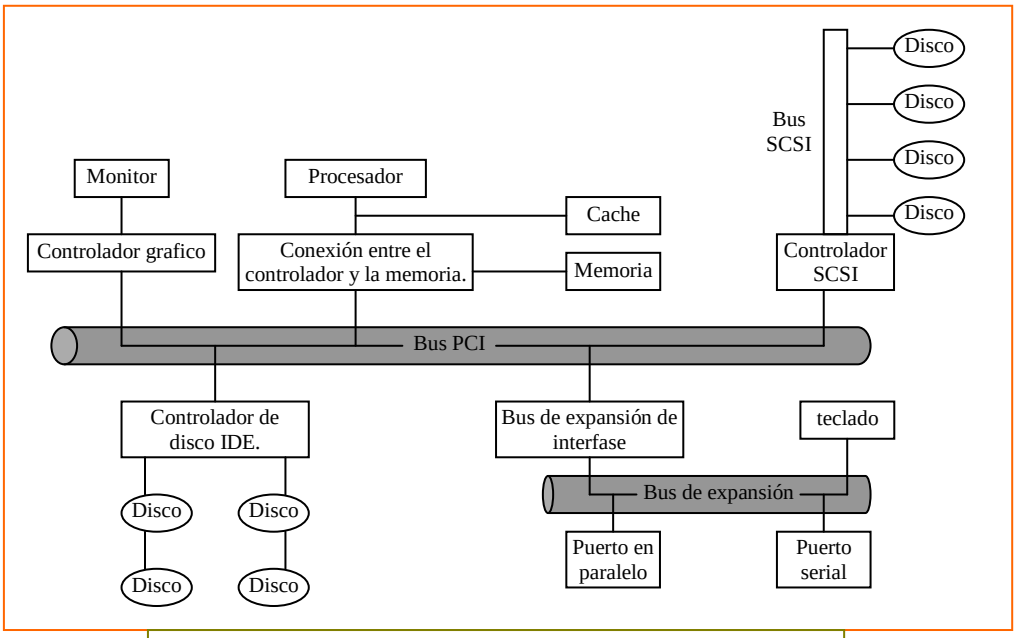


Figura 12.1 Una estructura típica de bus de una PC.

Un *controlador* es una colección de componentes electrónicos que pueden operar un puerto, un bus, o un dispositivo. Un controlador de puerto serial es un ejemplo de controlador de dispositivo simple. Es un único chip en la computadora que controla las señales sobre los cables de un puerto serial. En contraposición, un controlador de bus SCSI no es simple. Ya que el protocolo de SCSI es complejo, el controlador de bus SCSI es implementado como un circuito separado (un adaptador de host: *host adapter*) que se enchufa a la computadora. Éste típicamente contiene un procesador, micro código, y alguna memoria privada que le permite procesar los mensajes de protocolos SCSI. Algunos dispositivos tienen construidos sus propios controladores. Si se mira un drive de disco, se verá que tiene de un lado un circuito unido, el cual es el controlador de disco.
Para que el procesador pueda darle comandos y datos al controlador para conseguir la transferencia de I/O, el controlador tiene uno o más registros para datos y señales de control. El procesador se comunica con el controlador por medio de la lectura y escritura de patrones de bits en éstos registros. Una forma de que ocurra esta comunicación es a través del uso de instrucciones especiales de I/O que especifican la transferencia de un bit o palabra a una dirección de puerto de I/O. La instrucción de I/O activa las líneas de bus para seleccionar el dispositivo apropiado y mover los bits dentro y fuera del registro del dispositivo. Alternativamente, el controlador del dispositivo puede soportar el mapeo de memoria de I/O. En este caso, los registros de control del dispositivo son mapeados en el espacio de direcciones del

procesador. La CPU ejecuta los pedidos de I/O usando las instrucciones de transferencia de datos estándar para leer y escribir los registros de control del dispositivo.
Típicamente, un puerto de I/O consiste de cuatro registros llamados registros de **estado**, de **control**, **entrada de datos** y **salida de datos**. El registro de estado contiene bits que pueden ser leídos por el host. Estos bits indican los estados tales como si el comando actual se completo, si hay un byte listo en el registro de entrada de dato para ser leído, y si se ha producido un error en el dispositivo. El registro de control puede ser escrito por el host para comenzar un comando o para cambiar el modo de un dispositivo. Por ejemplo, un cierto bit en el registro de control de un puerto serial elige entre la comunicación totalmente duplex y la comunicación semi duplex, otro habilita el chequeo de paridad, un tercer bit setea el largo de una palabra a 7 u 8 bits, y otros bits seleccionan una de las velocidades soportadas por el puerto serial. El registro de entrada de dato es leído por el host para conseguir la entrada, y el registro de salida de datos es escrito por el host para sacar la información. Los registros de datos son típicamente desde 1 a 4 bytes. Algunos controladores tienen chips FIFO que pueden mantener varios bytes de entrada y salida para extender la capacidad del controlador más allá del tamaño de los registros de datos.

Polling: un completo protocolo para la interacción entre el host y un controlador puede ser confuso, pero la noción general es simple. Dicha noción se verá con un ejemplo. Asumamos que dos bits son usados para coordinar la relación productor-consumidor entre el controlador y el host. El controlador indica su estado a través del bit *busy* en el registro de estado (se recuerda que un bit seteado significa que esta en 1, mientras que un bit limpio esta en 0). El controlador setea el bit de busy cuando esta ocupado trabajando, y limpia el bit de busy cuando esta listo para aceptar el siguiente comando. El host señala su deseo por medio de un bit command-ready en el registro de comando. El host setea el bit command-ready cuando un comando esta disponible para que el controlador lo ejecute. Por ejemplo, el host saca los datos a través del puerto, coordinando con el controlador como sigue:

1. El host repetidamente lee el bit de busy hasta que se convierte en limpio.
2. El host setea el bit de escritura en el registro de comando y escribe un byte en el registro de salida de datos.
3. El host setea el bit command-ready.
4. Cuando el controlador nota que el bit command-ready esta seteado, éste setea el bit de busy.
5. El controlador lee el registro de comando y ve el comando escrito. Éste lee el registro de salida de datos para obtener el byte, y hace la I/O al dispositivo.
6. El controlador limpia el bit de command-ready, limpia el bit de error en el registro de estado para indicar que la I/O para el dispositivo fue exitosa, y limpia el bit de busy para indicar que ha finalizado.

Este loop es repetido para cada byte.
En el paso 1, el host esta ocupado esperando o polling: es decir, esta en loop, leyendo el registro de estado una y otra vez hasta que el bit de busy se convierte en limpio. Si el controlador y el dispositivo son rápidos, este método es bueno. Pero si la espera es larga, el host debería cambiar a otra tarea. Para saber cuando el controlador esta ocioso, para algunos dispositivos, el host debe servir al dispositivo rápidamente, o el dato será perdido. Por ejemplo, cuando los datos son recibidos desde un puerto serial o desde un teclado, el pequeño buffer en el controlador se revalsará y los datos se perderán si el host espera mucho tiempo antes de retornar a leer los bytes.
En muchas arquitecturas de computadoras, tres ciclos por instrucción en la CPU son suficientes para hacer poolling en un dispositivo: leer un registro de dispositivo, hacer un and-lógico para extraer el bit de estado, y branch si no es cero. Claramente, la operación básica de polling es eficiente. Pero polling se convierte ineficiente cuando es intentado repetidamente. En tales circunstancias, es más eficiente acordar que el hardware del controlador notifique a la CPU cuando el dispositivo esta listo para servir, en lugar de requerir que la CPU se fije repetidamente por la completitud de una I/O. El mecanismo de hardware que permite a un dispositivo notificar a la CPU es llamado interrupción.

Interrupciones: El mecanismo básico de interrupciones trabaja como sigue. El hardware de la CPU tiene un cable llamado línea de pedido de interrupción que la CPU percibe después de ejecutar cada instrucción. Cuando la CPU detecta que un controlador ha ingresado una señal en la línea de pedido de interrupción, la CPU almacena una cantidad de estados, tal como el actual valor del puntero de instrucción, y salta a la rutina *interrupt-handle* a una dirección fija de memoria. El manejador de interrupciones determina la causa de la interrupción, realiza el procedimiento necesario, y ejecuta una instrucción *return from interrupt* para retornar a la CPU al estado anterior de la interrupción. Se dice que el controlador del dispositivo forma una interrupción por medio del envío de una señal por la línea de pedido de interrupción, la CPU agarra la interrupción y se la envía al manejador de interrupciones, y el manejador atiende la interrupción sirviendo al dispositivo (Figura 12.3).

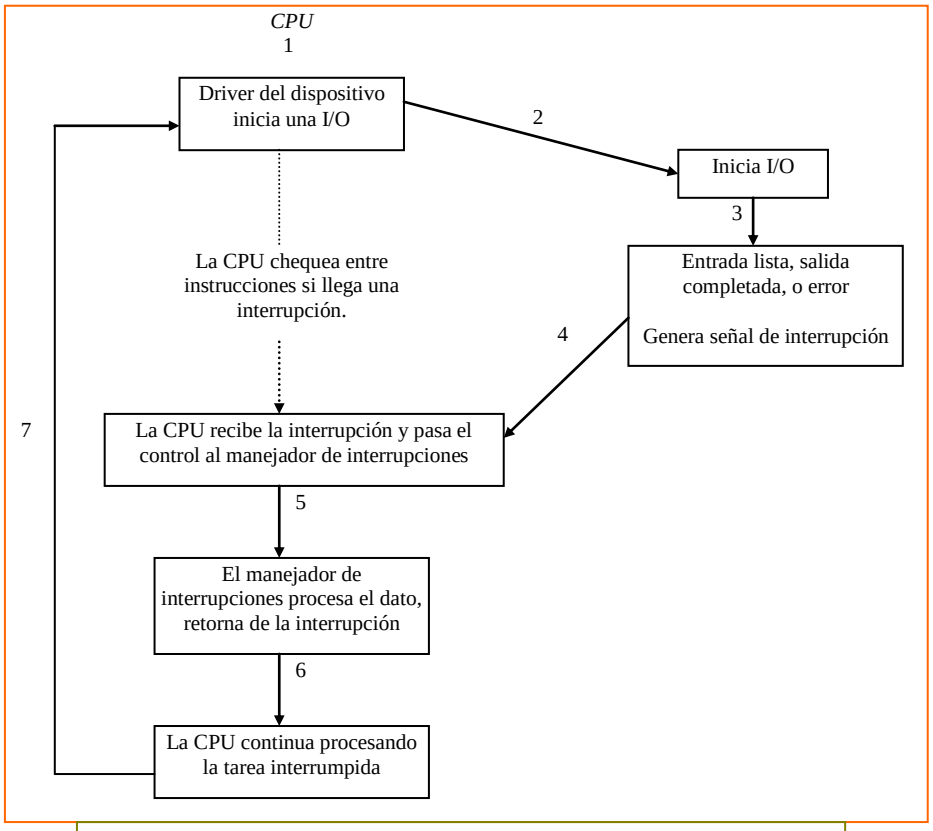


Figura 12.3 Un ciclo de un manejo de interrupción de I/O.

El mecanismo básico de interrupciones habilita a la CPU a responder a un evento asincrónico, tal como el controlador del dispositivo pase a estar listo para servir. En un sistema operativo moderno, se necesitan características más sofisticadas de manejo de interrupciones. Primero, se necesita la habilidad para retardar el manejador de interrupciones durante el procesamiento critico. Segundo, se necesita una forma eficiente de enviar la interrupción al manejador de interrupción para un dispositivo, sin primero preguntar a todos los dispositivos para averiguar cual fue el que produjo la interrupción. Tercero, se necesita un multinivel de interrupciones, para así el sistema operativo puede distinguir entre interrupciones de alta o baja prioridad, y así responder con el grado de urgencia apropiado. En modernos hardware de computadora, éstas tres características son proveídas por la CPU y por el hardware del controlador de interrupciones.

La mayoría de la CPUs tiene dos líneas de pedido de interrupciones. Una es para interrupciones de tipo *nonmaskable*, el cual es reservada para eventos tales como errores de memoria irrecuperables. La segunda línea de interrupción es para las de tipo *maskable*, tal como deshabilitar la CPU antes de la ejecución de una secuencia de instrucciones criticas que no deben ser interrumpidas. La interrupción de tipo maskable es usada por los controladores de los dispositivos para pedir servicio.

El mecanismo de interrupciones acepta una dirección (un número que selecciona una rutina especifica de manejo de interrupción de un pequeño conjunto). En la mayoría de las arquitecturas, ésta dirección es un offset en una tabla llamada vector de interrupción. Éste vector contiene las direcciones de memoria de las rutinas que manejan la interrupción. El propósito del mecanismo del vector de interrupciones es el de reducir la necesidad de que una única rutina de manejo de interrupciones busque todos los posibles orígenes de interrupciones para determinar cual es el servicio que se necesita. Sin embargo, en la practica las computadoras tiene más dispositivos (y ante esto, manejadores de interrupciones) que la cantidad de elementos que tiene el vector de interrupciones. Una forma común de resolver este problema es la de usar la técnica de cadena de interrupción, en el cual cada elemento en el vector de interrupciones apunta a la cabeza de una lista de manejadores de interrupciones. Al surgir una interrupción, los manejadores en la correspondiente lista son llamados uno por uno, hasta que se encuentre uno que pueda satisfacer el problema. Ésta estructura es un compromiso entre el overhead de una enorme tabla de interrupciones y la ineficiencia de un único manejador de interrupciones que despacha las interrupciones.
El mecanismo de interrupción también implementa un sistema de niveles de prioridad. Este mecanismo capacita a la CPU para diferir el manejo de interrupciones de baja prioridad con los de alta prioridad.
Un sistema operativo moderno interactúa con el mecanismo de interrupciones de varias maneras. Cuando el sistema arranca, el sistema operativo examina los buses para determinar cuales dispositivos están presentes, e instala los correspondientes manejadores de interrupciones en el vector de interrupción. Durante la I/O, las interrupciones surgen de varios controladores de dispositivos cuando están listos para servir. Estas interrupciones significan que la salida se ha completado, o que el dato entrante esta disponible, o que se ha detectado un fallo. El mecanismo de interrupción se utiliza también para manejar una gran variedad de excepciones; tales como dividir por cero, acceder a una dirección de memoria o protegida o no existente, o intentar ejecutar una instrucción privilegiada en modo usuario. Los eventos que disparan interrupciones tienen una propiedad en común: inducen a la CPU a ejecutar una rutina urgente.
Un sistema operativo tiene otros buenos usos para que un mecanismo de hardware eficiente almacene una pequeña cantidad de estados del procesador, y luego llama a una rutina privilegiada en el kernel. Por ejemplo, muchos sistemas operativos usan el mecanismo de interrupciones para el paginado de memoria virtual. Un fallo de página es una excepción que provoca una interrupción. La interrupción suspende el proceso actual y salta al manejador de fallo de página en el kernel. Este manejador almacena el estado del proceso, mueve el proceso a la cola de espera, realiza la administración de la página, produce una operación de I/O para conseguir la página, elige un nuevo proceso para continuar su ejecución, y luego retorna de la interrupción.
Otro ejemplo es la implementación de llamadas al sistema. Una llamada al sistema es una función que es llamada por una aplicación que invoca un servicio del kernel. La llamada al sistema chequea los argumentos que le dio la aplicación, construyendo una estructura para transportar estos argumentos al kernel, y luego ejecuta una instrucción especial llamada una *interrupción de software o trap*. Esta instrucción tiene un operando que identifica el servicio de kernel deseado. Cuando la llamada al sistema ejecuta la instrucción de trap, el hardware de interrupción almacena el estado del código del usuario, cambia el modo del supervisor, y salta a la rutina del kernel que implementa el servicio pedido. El trap tiene una prioridad más baja que las prioridades de las interrupciones de dispositivos (ejecutar una llamada al sistema en medio de una aplicación es menos urgente que servir a un controlador de dispositivo antes de que la cola FIFO se revalse y se pierda el dato).
Las interrupciones pueden también ser usadas para manejar el flujo de control en el kernel. Por ejemplo, consideremos el procesamiento requerido para completar una lectura de disco. Un paso es el de copiar el dato desde el espacio del kernel al espacio del buffer del usuario. Esta copia consume tiempo, pero no es urgente (es decir, no debería bloquear al manejador de interrupciones de alta prioridad). Otro paso seria el de comenzar la siguiente I/O pendiente para este drive de disco. Este paso tiene una alta prioridad: si el disco no esta siendo usado eficientemente, se necesita comenzar la siguiente I/O tan pronto como la anterior I/O es completada. Consecuentemente, el código del kernel que completa una lectura de disco esta implementada por un par de manejadores de interrupciones. El manejador de alta prioridad registra el estado de la I/O, anula el dispositivo de interrupciones, comienza la siguiente I/O pendiente, y

promueve una interrupción de baja prioridad para completar el trabajo. Luego, cuando la CPU no es ocupada con trabajos de alta prioridad, la interrupción de baja prioridad será servida. El correspondiente manejador completa la I/O a nivel de usuario copiando los datos desde los buffers del kernel al espacio de la aplicación, y luego llama al scheduler para ubicar la aplicación en la cola de listos.
En Solaris, los manejadores de interrupciones son ejecutados como thread kernel. Un rango de altas prioridades es reservada para estos threads. Estas prioridades dan a los manejadores de interrupciones precedencia sobre los códigos de aplicaciones y otros códigos del kernel, e implementan las relaciones de prioridad entre los manejadores de interrupciones. Las prioridades causan al scheduler de thread de Solaris desalojar a los manejadores de interrupciones de baja prioridad en favor de los de alta prioridad, y la implementación de thread permite a un hardware multiprocesador correr varios manejadores de interrupciones concurrentemente.
En resumen, las interrupciones son usadas en modernos sistemas operativos para manejar eventos asincrónicos y para atrapar las rutinas en modo supervisor en el kernel. Para que el trabajo más urgente sea el que se haga primero, las computadoras modernas usan un sistema de prioridades de interrupciones. Los controladores de dispositivo, fallas de hardware, y llamadas al sistema emiten interrupciones que activan rutinas en el kernel.

Acceso directo a memoria: Para un dispositivo que realiza grandes transferencias, tal como un drive de disco, es muy gastador usar un caro procesador de propósito general para que este mirando continuamente los bits de estado y para que alimente con datos a un registro del controlador con solo 1 byte a la vez (proceso llamado I/O programada: PIO). Muchas computadoras evitan agobiar la CPU principal con PIO por medio de que éste trabajo lo realice un procesador de propósito especial llamado controlador de acceso directo a memoria (DMA). Para inicial una transferencia DMA, el host escribe un bloque de comando DMA en la memoria. Este bloque contiene un puntero al origen de una transferencia, un puntero al destino de la transferencia, y la cantidad de bytes a ser transferidos. La CPU escribe la dirección de este bloque de comando en el controlador de DMA, y sigue con otro trabajo. Entonces, el controlador de DMA procede para operar el bus de memoria directamente, ubicando direcciones sobre el bus para realizar la transferencia sin la ayuda de la CPU principal. Un simple controlador de DMA es un componente estándar en PCs.
La comunicación entre el controlador de DMA y el controlador del dispositivo se realiza vía un par de líneas (cables) llamadas *DMA-request* (pedido de DMA), y *DMA-acknowledge* (reconocedor de DMA). El controlador del dispositivo ubica una señal en la línea DMA-request cuando una palabra de datos esta lista para ser transferida. Esta señal causa que el controlador de DMA se apodere del bus de memoria, ponga la dirección deseada en las líneas de dirección de memoria, y ponga una señal en la línea de DMA-ackonwledge. Cuando el controlador del dispositivo recibe la señal del DMA-acknowledge, transfiere la palabra de dato a memoria, y elimina la señal en el DMA-request. Cuando se realiza la trasferencia completa, el controlador de DMA interrumpe la CPU (Figura 12.5).

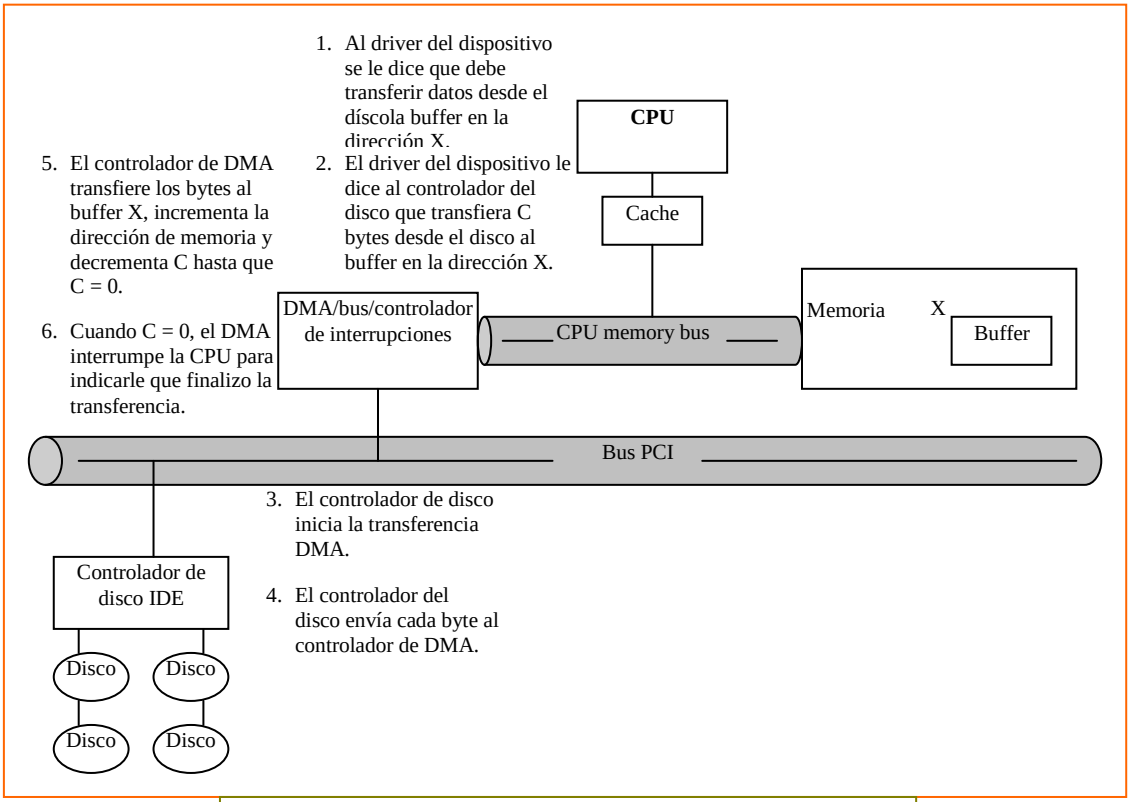


Figura 12.5 Pasos en una transferencia DMA.

Note que, cuando el controlador de DMA se apodera del bus de memoria, la CPU es momentáneamente prevenida de acceder a la memoria principal, aunque la CPU pueda acceder a los items de datos en su cache primaria y secundaria. Aunque este *cicle stealing* (ciclo robante) puede hacer más despacio el cálculo de la CPU, manejar la transferencia de datos por medio del controlador DMA generalmente aumenta la performance global del sistema. Algunas arquitecturas de computadoras usan direcciones de memoria físicas para DMA, pero otras realizan el acceso a memoria virtual directamente (DVMA: *direct virtual memory access*), usando direcciones virtuales que son sometidas a una traducción desde una dirección de memoria virtual a una dirección de memoria física. DVMA pueden realizar una transferencia entre dos dispositivos que usan mapeo de memoria sin la intervención de la CPU o la utilización de la memoria principal.

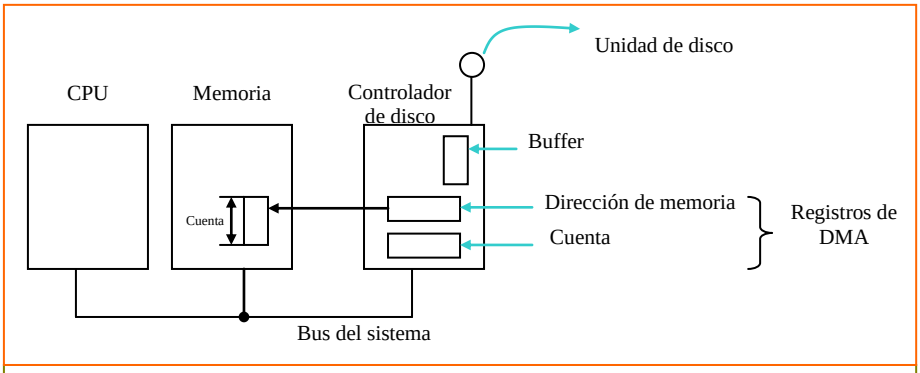
Sobre el kernel en modo protegido, el sistema operativo generalmente previene a los procesos emitir directamente comandos a los dispositivos. Esta disciplina protege a los datos de violaciones de control de accesos, y también protege al sistema de erróneos usos de los controladores de dispositivos que pueden causar una caída del sistema. En lugar de esto, el sistema operativo exporta funciones que un proceso suficientemente privilegiado puede usar para acceder a las operaciones de bajo nivel sobre el subyacente hardware. En kernels sin protección de memoria, los procesos pueden acceder a los controladores de los dispositivos directamente. Este acceso directo puede ser usado para obtener una alta performance, ya que esto puede evitar la comunicación con el kernel, context switch, y capas de software del kernel. Desafortunadamente, esto se interfiere con la seguridad del sistema y la estabilidad. La tendencia en los sistemas operativos de propósito general es la de proteger la memoria y los dispositivos.

Aunque los aspectos de hardware de I/O son complejos se consideran a un nivel de detalle como el de los diseñadores de hardware electrónico, los conceptos que se describieron son suficientes para entender muchos aspectos de I/O del sistema operativo. Hagamos una revisión de los conceptos:

- Un bus.
- Un controlador.
- Un puerto de I/O y sus registros.

- Acuerdos de la relación entre el host y el controlador del dispositivo.
- La ejecución de estos acuerdos en un loop de intercambio (polling) o vía interrupciones.
- La realización de este trabajo por medio de un controlador de DMA para grandes transferencias.

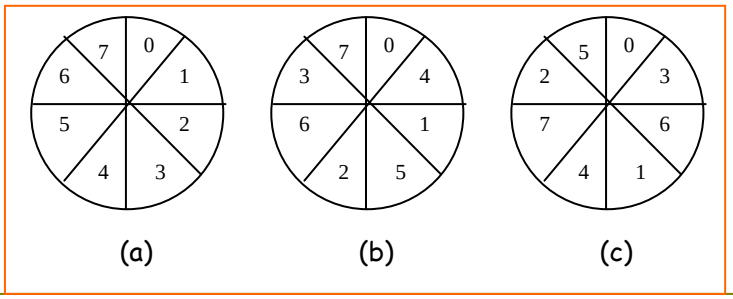
Acceso directo a memoria (Tanenbahum): muchos controladores, sobre todos los dispositivos por bloques, manejan acceso *directo a memoria* o *DMA*. Parra explicar el funcionamiento de DMA, veamos primero como ocurren las lecturas a disco cuando no se usa DMA. Primero, el controlador lee el bloque (uno o más sectores) de la unidad en serie, bit por bit, hasta que todo el bloque esta en el buffer interno del controlador. A continuación, el controlador calcula la suma de verificación para comprobar que no ocurrieron errores de lectura, y luego causa una interrupción. Cuando el sistema operativo comienza a ejecutarse, puede leer el bloque del disco del buffer del controlador byte por byte o palabra por palabra, ejecutando un loop, leyendo en cada iteración un byte o una palabra de un registro del controlador y almacenándose en la memoria.
Naturalmente, un ciclo de la CPU programado para leer los bytes del controlador uno por uno desperdicia tiempo de CPU. Se invento el DMA para liberar a la CPU de este trabajo de bajo nivel. Cuando se usa DMA, la CPU proporciona al controlador dos elementos de información, además de la dirección en disco del bloque: la dirección de memoria donde debe colocarse el bloque, y el número de bytes que deben transferirse, como se muestra en la figura de abajo.



Una transferencia de DMA es realizada totalmente por el controlador.

Una vez que el controlador ha leído todo el bloque del dispositivo, lo ha colocado en su buffer y ha calculado la suma de verificación, copia el primer byte o palabra en la memoria principal en la dirección especificada por la dirección de memoria de DMA. Luego, el controlador incrementa la dirección de DMA y decrementa la cuenta de DMA en el número de bytes que se acaban de transferir. Este proceso se repite hasta que el registro cuenta es cero, y en ese momento el controlador causa una interrupción. Cuando el sistema operativo inicia, no tiene que copiar el bloque en la memoria; ya está ahí.
Pero, ¿por qué el controlador no almacena los bytes en la memoria principal tan pronto como los recibe del disco?, ¿por qué es que necesita un buffer interno?. La razón es que una vez que se ha iniciado una transferencia de disco, los bits siguen llegando del disco a velocidad constante, sea que el controlador esté o no listo para recibirlos. Si el controlador tratara de escribir los datos directamente en la memoria, tendría que hacerlo a través del bus del sistema para cada palabra transferida. Si el bus estuviese ocupado porque otro dispositivo lo está usando, el controlador tendría que esperar. Si la siguiente palabra del disco llegara antes que la anterior se almacene en memoria, el controlador debe ponerla en algún lado. Si el bus estuviese muy ocupado, el controlador debería tener que almacenar una gran cantidad de palabras, y también realizar un gran número de tareas administrativas. Si el bloque se guarda en el buffer interno, no se necesitará el bus en tanto no se inicie el DMA, y el diseño del controlador será mucho más sencillo porque la transferencia DMA a memoria no depende críticamente del tiempo. De hecho, algunos controladores viejos sí transferían directamente a memoria con un mínimo de almacenamiento intermedio interno, pero cuando el bus estaba muy ocupado, a veces era necesario terminar una transferencia con un error de desbordamiento.

Un proceso con almacenamiento intermedio de dos pasos como acabamos de describir tiene implicaciones importantes para el rendimiento de I/O. Mientras los datos están siendo transferidos desde el controlador a la memoria, sea por la CPU o por el controlador, el siguiente sector estará pasando bajo la cabeza del disco y los bits estarán llegando al controlador. Los controladores sencillos simplemente no pueden efectuar entrada y salida al mismo tiempo, de modo que cuando se está realizando una transferencia a la memoria, el sector que pasa bajo la cabeza del disco se pierde.
En consecuencia, el controlador sólo puede leer bloques de manera intercalada, es decir, uno sí y uno no, de modo que la lectura de todo una pista requiere dos rotaciones completas, una para los bloques pares y otra para los bloques impares. Si el tiempo que toma transferir un bloque del controlador a la memoria por el bus es más largo que el que toma leer un bloque de disco, puede ser necesario leer un bloque y luego saltar dos (o más) bloques.
Saltar bloques para dar al controlador tiempo de transferir los datos a la memoria se denomina **intercalación**. Cuando se da formato al disco, los bloques se numeran teniendo en cuenta el factor de intercalación. En la figura (a) se ve un disco con ocho bloques por pista y cero intercalación. En la (b) se ve el mismo disco con intercalación sencilla. En la (c) se ve la intercalación doble.



(a) Sin intercalación. (b) Intercalación sencilla. (c) Intercalación doble.

No todas las computadoras usan DMA. El argumento en su contra es que en muchos casos la CPU principal es mucho más rápida que el controlador de DMA y puede realizar el trabajo en mucho menos tiempo. Si la CPU (rápida) no tiene otra cosa que hacer, obligarla a esperar hasta que el controlador de DMA (lento) termine no tiene sentido. Además, si se omite el controlador de DMA y se deja que la CPU realice todo el trabajo, se ahorra dinero.

Aplicación de la interface de I/O

En esta parte se discutirán las técnicas de estructuras e interfaces para que se le permita al sistema operativo tratar a los dispositivos de I/O de una manera estándar e uniforme. Se verá como una aplicación puede abrir un archivo en un disco sin el conocimiento de que tipo de disco es, y como los nuevos discos y otros dispositivos se pueden agregar a la computadora sin la necesidad de romper el sistema operativo.
El problema de la necesidad de que los dispositivos sean tratados de una manera estándar involucra abstracción, encapsulamiento, y software modelado en capas. Especialmente, podemos abstraer más allá de los diferentes detalles que tienen los dispositivos de I/O por la identificación de unos pocos tipos. Cada uno de estos tipos generales es accedido por medio de un conjunto estándar de funciones: una *interfase*. Las actuales diferencias están encapsuladas en módulos del kernel llamados *drivers de* dispositivos (device *drivers*), que son hábitos que cada dispositivo puede tolerar. La figura 12.6 muestra como las porciones de relaciones de I/O del kernel están estructuradas en un software de capas.

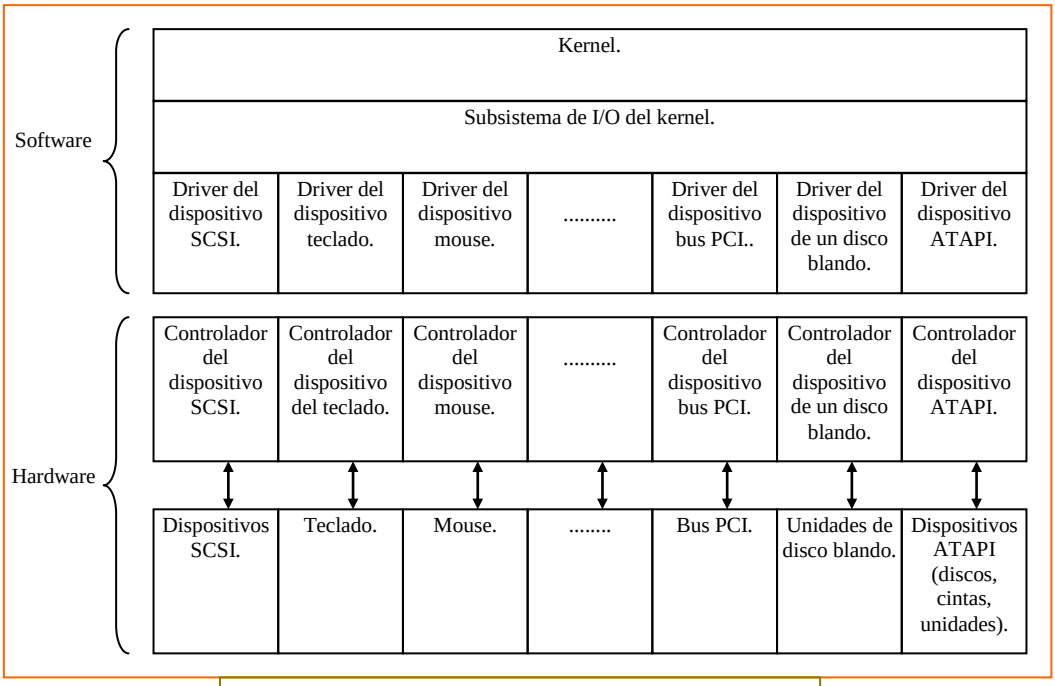


Figura 12.6 Estructura de I/O del kernel.

El propósito de la capa del driver del dispositivo es el de esconder las diferencias entre los controladores del dispositivo y el subsistema de I/O del kernel, así como las llamadas del subsistema de I/O encapsula el comportamiento de los dispositivos en una pocas clases genéricas que esconden las diferencias entre el hardware y las aplicaciones. Haciendo el subsistema de I/O independiente del hardware simplifica el trabajo tanto del diseñador del sistema operativo como de los fabricantes del hardware. Ambos diseñan nuevos dispositivos para que sean compatibles con la interfase de un controlador de host (tal como SCSI-2), o escriban nuevos drivers de dispositivo para unir el nuevos hardware a un sistema operativo existente. Así, los nuevos dispositivos pueden ser unidos a una computadora sin la necesidad de esperar que un fabricante de sistemas operativos desarrolle uno con código de soporte. Desafortunadamente para los fabricantes del hardware del dispositivo, cada tipo de sistema operativo tiene sus propios estándares para la interfase del driver del dispositivo. Un dispositivo dado se puede trasportar con múltiples drivers de dispositivos, por ejemplo, drivers para MS-DOS, Windows 95, Windows NT, y Solaris.
Los dispositivos varían en muchas dimensiones:

- *Bloque o flujo de caracteres:* un dispositivo de flujo de caracteres transfiere uno a uno los bytes, mientras que un dispositivo de bloque transfiere un bloque de bytes como una unidad.
- *Acceso secuencial o aleatorio:* un dispositivo de acceso secuencial transfiere datos en un orden fijo el cual es determinado por el dispositivo, mientras que el usuario de un dispositivo de acceso aleatorio puede instruir al dispositivo para buscar en cualquier ubicación del almacenamiento.
- *Sincrónico o asincrónico:* un dispositivo sincrónico es aquel que realiza la transferencia de datos con tiempos de respuestas predecibles. Un dispositivo asincrónico exhibe tiempos de respuestas irregulares o impredecibles.
- *Compartidos o privados (dedicados):* un dispositivo compartido puede ser usado concurrentemente por varios procesos o threads; un dispositivo dedicado no puede.
- *Velocidad de operación:* la velocidad de los dispositivos esta en un rango entre unos pocos bytes por segundo hasta unos pocos gigabytes por segundo.
- *Lectura-escritura, solo lectura, solo escritura:* algunos dispositivos realizan tanto entrada como salida, pero otros soportan solo una dirección de datos.

Dispositivos de block y de caracteres: los dispositivos de I/O se pueden dividir en dos tipos: *dispositivos por bloques* y *dispositivos por caracteres*. Un dispositivo por bloques almacena información en

bloques de tamaño fijo, cada uno con su propia dirección. Los tamaños de bloques comunes van desde 512 bytes hasta 32 Kb (32768 bytes). La propiedad esencial en un dispositivo por bloques es que es posible leer o escribir cada bloque con independencia de los demás. Los discos son los dispositivos por bloques más comunes.
Otro tipo de dispositivos de I/O es el dispositivo por caracteres. Un dispositivo de este tipo suministra o acepta un stream de caracteres, sin contemplar ninguna estructura de bloques; no es direccionable y no tiene una operación de búsqueda. Las impresoras, interfaces de red, mouses y caso todos los demás dispositivos que no se parecen a los discos pueden considerarse como dispositivos por caracteres (algunos dispositivos, tal como los relojes, no entran en ninguno de éstos dos tipos).
La interfase de un dispositivo de block captura todos los aspectos necesarios para acceder a los drive de disco y a otros dispositivos orientados a bloques. Se espera que el dispositivo entienda los comandos tales como read y write, y, si el dispositivo es de acceso aleatorio, tiene un comando seek para especificar cual bloque es el siguiente a transferir. Las aplicaciones normalmente acceden a tales dispositivos por medio de la interface del sistema de archivos. Los sistemas operativos, y aplicaciones especiales como la administración de un sistema de base de datos, pueden preferir acceder a un dispositivo de bloque como un simple arreglo lineal de bloques. Este modo de acceso es a veces llamada *raw I/O*. Se puede ver que write, read y seek involucran el comportamiento esencial de los dispositivos de almacenamiento de bloques, por lo que las aplicaciones son aisladas de las diferencias de bajo nivel que tienen éstos dispositivos.
El acceso a archivo por medio del mapeo a memoria puede ser dividido en capas. En lugar de ofrecer operaciones de lectura-escritura, la interfase de mapeo de memoria provee acceso al almacenamiento de disco por medio de la ubicación de un arreglo de bytes en memoria. La llamada al sistema que mapea un archivo en memoria retorna la dirección de memoria virtual de un arreglo de caracteres que contiene una copia del archivo. La transferencia del dato actual es realizada solo cuando se necesita satisfacer un acceso a la imagen de memoria. Ya que las transferencias son manejadas por el mismo mecanismo como el usado en el acceso a memoria virtual de paginado por demanda, el mapeo de memoria de la I/O. El paginado de la memoria es también conveniente para programadores: el acceso al archivo por mapeo de memoria es tan simple como leer y escribir en memoria. Esto es común para sistemas operativos que ofrecen memoria virtual para usar la interfase de mapeo para servicios del kernel. Por ejemplo, para ejecutar un programa, el sistema operativo mapea el ejecutable en memoria, y luego transfiere el control a la dirección entrante del ejecutable.
Un teclado es un ejemplo de dispositivo que es accedido a través de una interfaz de stream de caracteres. Las llamadas al sistema básicas en esta interfaz permiten a una aplicación sacar o meter un carácter. Sobre esta interfaces se pueden construir librerías que ofrecen un acceso en tiempo de línea, con servicios de buffer y edición (por ejemplo, cuando un usuario teclea un retroceso, el carácter precedido es eliminado de stream entrante). Este estilo de acceso es conveniente para dispositivos tales como teclados, mouse, y módems, el cual producen datos para ser ingresados instantáneamente, es decir, a tiempos que no pueden necesariamente ser predecidos por la aplicación. Este estilo de acceso es bueno también para dispositivos de salida tales como impresoras o parlantes.

Dispositivos de red: ya que las características de performance y rendimiento de una I/O de red difieren significativamente de aquellas I/O de disco, la mayoría de los sistemas operativos proveen una interfase de I/O de red que es diferente de la interfase *read-write-seek* usada para discos. Una interfase que esta disponible en muchos sistemas operativos, incluyendo UNIX y Windows NT, es la interfase de red socket (enchufe). Piense en un enchufe de electricidad de pared: cualquier aparato domestico puede ser enchufado. Por analogía, las llamadas al sistema en la interfase socket permite a una aplicación crear un enchufe, para conectar un socket local a una dirección remota (el cual enchufa esta aplicación en un enchufe creado por otra aplicación), para escuchar por cualquier aplicación remota enchufada en el socket local, y enviar y recibir paquetes sobre la conexión. Para soportar la implementación de servidores, la interfase socket también provee una función llamada *select* que administra un conjunto de sockets. Una llamada a select retorna información sobre cual socket tiene un paquete esperando para ser recibido, y cuales socket tienen lugar para aceptar un paquete a ser enviado. El uso de select elimina el polling y

espera ocupada que, de otra manera, habrían sido necesarias para I/O de red. Estas funciones encapsulan el comportamiento de redes, facilitando de gran manera la creación de aplicaciones distribuidas que pueden usar cualquier hardware de red fundamental y protocolo de pila.

Relojes y timers: la mayoría de las computadoras tiene relojes y timers (hardware) que proveen tres funciones básicas:

- Dan la hora actual.
- Dan el tiempo transcurrido.
- Setean un timer para activar la operación X en el tiempo P.

Estas operaciones son muy usadas por el sistema operativo, y también por aplicaciones basadas en el tiempo. Desafortunadamente, las llamadas al sistema que implementan éstas funciones no son estándares entre los diferentes sistemas operativos.
El hardware para medir el tiempo y activar operaciones es llamado *programmable interval timer*. Este puede ser seteado para esperar una cierta cantidad de tiempo y luego generar una interrupción. Este puede ser seteado para hacer ésta operación una vez, o para que repita el proceso, generando interrupciones periódicas. Este mecanismo es usado por el scheduler para generar una interrupción que dará fina a un proceso de su parte de uso de CPU. Este es usado por el subsistema de I/O para vaciar los buffers de cache sucios, y por el subsistema de red para cancelar operaciones que son demasiado lentas ya sea por la congestión de la red o por fallas. El sistema operativo puede proveer también una interfase para que procesos del usuario usen timers. El sistema operativo puede soportar más pedidos de timer que el número de timers (en cuanto al hardware) simulando relojes virtuales. Para hacerlo, el kernel (o el driver del dispositivo del timer) mantiene una lista de interrupciones deseadas por sus propias rutinas y por pedidos del usuario, ordenándola de menor a mayor según el momento que debe ocurrir la interrupción. Este setea el timer para el primer tiempo. Cuando el timer interrumpe, el kernel le avisa al pedidor, y recarga el timer con el siguiente tiempo (es decir, el siguiente elemento de la lista).
En muchas computadores, los costos de interrupción generados por el tic-tac del reloj (hardware) son entre 18 y 60 tic-tacs por segundo. Esta resolución es grosera, ya que una computadora moderna puede ejecutar sientas de millones de instrucciones por segundo. La precisión de las activaciones es limitada por la grosera resolución del timer, juntado con el overhead de mantener relojes virtuales. Y, si además el sistema de tic-tac es usado para mantener el reloj de la maquina, el reloj puede que este sin rumbo. En la mayoría de las computadoras, el hardware de reloj es construido a partir de una alta frecuencia de contadores.

I/O bloqueante y no bloqueante: Un aspecto que queda de las llamadas al sistema es el que se elige entre I/O bloqueante y no bloqueante (asincrónica). Cuando una aplicación emite una llamada al sistema *blocking*, la ejecución de la aplicación es suspendida. La aplicación es movida a una cola de espera. Luego de que ésta llamada se completa, la aplicación es movida nuevamente a la cola de listos, donde será elegida para continuar su ejecución, y en ese momento recibirá los valores retornados por la llamada al sistema.
Algunos procesos a nivel de usuario necesitan I/O no bloqueante. Un ejemplo es una interfase de usuario que recibe la entrada desde el teclado y el mouse mientras procesa y muestra datos sobre la pantalla. Otro ejemplo es una aplicación de video que lee frames de un archivo en disco mientras simultáneamente descomprime y muestra la salida en pantalla.
Una forma que tiene una aplicación para solapar su ejecución con la I/O es la de escribir una aplicación multithread. Algunos thread pueden realizar llamadas al sistema bloqueantes, mientras otros continúan su ejecución. Los desarrolladores de Solaris usaron esta técnica para implementar una librería a nivel de usuario para la I/O asincrónica, librando al escritor de la aplicación de esta tarea. Algunos sistemas operativos proveen llamadas al sistema de I/O no bloqueante. Una llamada no bloqueante no para la ejecución de la aplicación por un largo tiempo. En lugar de esto, esta retorna rápidamente, con un valor de retorno que indica que cantidad de bytes fueron transferidos.

Una alternativa a una llamada al sistema no bloqueante es una llamada al sistema asincrónica. Una llamada al sistema asincrónica retorna inmediatamente, sin esperar que la I/O se complete. La aplicación continua ejecutando su código, y la completitud de la I/O en el futuro se comunica a la aplicación, ya sea por medio del seteo de una variable en el espacio de dirección de la aplicación, o a través de la activación de una señal o una interrupción de software . Note la diferencia entre llamadas al sistema no bloqueantes y llamadas al sistema asincrónicas. Los read no bloqueantes retorna inmediatamente con cualquier dato que este disponible (el número total de bytes pedidos, menos, o ninguno en absoluto). Una llamada read asincrónica pide una transferencia que será realizada en su totalidad, pero será realizada en el futuro.

Subsistema de I/O del kernel

El kernel provee muchos servicios relacionados con la I/O. En esta parte, se describirán varios servicios que son proveídos por el subsistema de I/O del kernel. Los servicios que se verán son scheduling de I/O, buffering, caching, spooling, reservación de dispositivos, y manejo de errores

Scheduling de I/O: Planificar un conjunto de pedidos de I/O significa determinar un buen orden en el cual ejecutarlos. El orden en el cual las aplicaciones emiten llamadas al sistema raramente es la mejor elección. La planificación puede mejorar la performance global del sistema, puede distribuir los accesos a los dispositivos de manera justa, y puede reducir el tiempo de espera promedio para completar una I/O. Veamos un ejemplo. Supongamos que el brazo del disco esta cerca del comienzo del disco, y que tres aplicaciones invocan un llamado de lectura al disco. La aplicación 1 pide un bloque cerca del final del disco, la aplicación 2 pide uno cerca del inicio, y la aplicación 3 pide uno en el medio. Esta claro que el sistema operativo puede reducir la distancia que el brazo del disco recorre sirviendo a las aplicaciones en orden 2, 3, 1. El trabajo de ordenar la secuencia de pedidos de esta forma es el trabajo del scheduler. Los desarrolladores de sistemas operativos implementan la planificación de la cola de los pedidos de cada dispositivo. Cuando una aplicación invoca una llamada de I/O bloqueante, el servicio es ubicado en la cola de este dispositivo. El scheduler de I/O reorganiza el orden de la cola para mejorar la eficiencia global del sistema y el tiempo de respuesta promedio esperado por las aplicaciones. El sistema operativo trata también de ser justo, para que una aplicación no reciba un servicio pobre, o le dé prioridad a servicios provocando que se retrasen otros. Por ejemplo, los pedidos del subsistema de memoria virtual pueden tomar prioridad sobre los pedidos de las aplicaciones.
Una forma de que el subsistema de I/O mejore la eficiencia de la computadora es planificando las operaciones de I/O. Otra forma es usando espacio de almacenamiento en la memoria principal o en disco, por medio de técnicas llamadas buffering, caching, y spooling.

Buffering: un buffer es un área de memoria que almacena datos mientras ellos son transferidos entre dos dispositivos o entre un dispositivo y una aplicación. El buffering es realizado por tres razones. Una razón es cubrir con una desigual velocidad entre el productor y el consumidor del dato. Supongamos, por ejemplo, que un archivo esta siendo recibido vía modem para ser almacenado en disco duro. El modem es aproximadamente miles de veces más lento que el disco duro. Por lo que un buffer es creado en memoria principal para acumular bytes que son recibidos desde el modem. Cuando se ha recibido un buffer completo, el buffer puede ser escrito en disco en una única operación. Ya que la escritura en el disco no es instantánea y el modem necesita aun un espacio para almacenar los datos que le van llegando, se utilizan dos buffers. Luego de que el modem llena el primer buffer, se pide una escritura de disco. El modem empieza entonces a llenar el segundo buffer mientras el primer buffer es escrito en el disco. Cuando el modem llena el segundo buffer, la operación de escritura en el disco del primer modem se debe de haber completado por lo que el modem cambia y comienza a escribir en el primer buffer, empezándose a escribir a disco el segundo buffer. Este doble buffering desacopla al productor de datos del consumidor, relajando los requerimientos de tiempo entre ellos.
Un segundo uso de buffering es para ajustar dispositivos que tienen diferentes tamaños de transferencia de datos. Tales disparidades son muy comunes en redes de computadoras, donde los buffers son usados

para la fragmentación y reensamble de mensajes. Por el lado del enviador, un gran mensaje es fragmentado en pequeños mensajes. Los paquetes son enviados por la red, y el lado del recibidor los ubica en un buffer para reensamblarlos y formar un único mensaje.
Un tercer uso es para soportar copia semántica para aplicaciones de I/O. Un ejemplo aclarara la idea de "copia semántica". Supongamos que una aplicación tiene un buffer de datos que desea escribir en disco. Ésta invoca la llamada al sistema write, proporcionando un puntero al buffer, y un entero especificando la cantidad de bytes a copiar. Luego de que la llamada al sistema retorna pero la copia no se ha llevado a cabo, si la aplicación cambia el contenido del buffer puede que en el disco se copie una versión del dato errónea. Para evitar esto se utiliza la copia semántica, donde la versión del dato escrito en disco es la versión del dato que estaba en el momento en que la aplicación invoco la llamada write, independientemente de cualquier subsecuente cambio en el buffer de la aplicación. Una forma simple de que el sistema operativo pueda garantizar la copia semántica es que la llamada al sistema write copie los datos de la aplicación en un buffer del kernel antes de que retorne el control a la aplicación. La escritura del disco es realizada con los datos que están en el buffer del kernel, por lo que los subsecuentes cambios en el buffer de la aplicación no afectaran. Copiar los datos entre buffers del kernel y el espacio de datos de la aplicación es común en sistemas operativos, a pesar del overhead que ésta operación introduce. El mismo efecto puede ser obtenido de manera más eficiente por el inteligente uso del mapeo de la memoria virtual.

Caching: Un *cache* es una región de memoria rápida que mantiene copia de datos. Los accesos a las copias de datos de la cache son más eficientes que los accesos a la original. Por ejemplo, las instrucciones del actual proceso en ejecución están almacenadas en disco, guardadas también en la memoria física, y copiadas nuevamente en las caches primaria y secundaria de la CPU. La diferencia entre un buffer y una cache es que un buffer puede mantener la copia solo de un dato existente, mientras que una cache, por definición, solo mantiene una copia en almacenamiento rápido de un ítem que puede estar residiendo en cualquier parte.
Caching y buffering son dos funciones distintas, pero a veces una región de memoria puede ser usada tanto para buffering como para caching. Por ejemplo, para preservar las copias semánticas y permitir scheduling de I/O de disco eficientes, el sistema operativo usa buffers en memoria para mantener los datos del disco. Estos buffers son también usados como cache, para mejorar la eficiencia de la I/O para archivos que están compartidos por aplicaciones, o que están siendo escritos o releídos rápidamente. Cuando el kernel recibe un pedido de I/O de un archivo, el kernel primero accede al buffer de la cache para ver si la región del archivo ya esta en la memoria principal. Si está, una I/O física al disco puede ser evitada o retrazada. Asimismo, las escrituras de disco son acumuladas en el buffer de la cache por varios segundos, por lo que las grandes transferencias son amontonadas para permitir schedulers de escritura eficientes.

Reserva de dispositivos y spooling: Un spool es un buffer que mantiene la salida para un dispositivo, tal como una impresora, que no puede aceptar streams de datos intercalados. Aunque una impresora puede realizar solo un trabajo a la vez, varias aplicaciones pueden desear imprimir sus aplicaciones concurrentemente, sin tener sus salidas todas mezcladas. El sistema operativo resuelve este problema interceptando todas las salidas a la impresora. Cada salida de las aplicaciones es spooled a un archivo de disco separado. Cuando una aplicación finaliza la impresión, el sistema de spooling encola el correspondiente archivo spool y es enviado a la impresora. El sistema de spooling copia los archivos encolados en el spool desde el spool a la impresora uno a la vez.
Algunos dispositivos, tales como drivers de cinta e impresoras, no pueden multiplexar útilmente pedidos de I/O de múltiple aplicaciones concurrentes. Spooling es una forma en que el sistema operativo puede coordinar la salida concurrente. Otra forma de tratar con dispositivos de acceso concurrente es proveer facilidades explícitas para la coordinación. Algunos sistemas operativos (incluyendo VMS) proveen soporte para el acceso exclusivo a dispositivos, permitiendo a un proceso asignarle un dispositivo ocioso, y desasignar el dispositivo cuando ya no lo use más. Otros sistemas operativos proveen funciones que permiten a los procesos coordinar el acceso exclusivo entre ellos. Por ejemplo, Windows NT provee llamadas al sistema para esperar hasta que un dispositivo se convierta disponible. Este también tiene un

parámetro para la llamada al sistema open que declara el tipo de acceso que le será permitido a los otros threads concurrentes.

Manejo de errores: un sistema operativo que usa memoria protegida puede custodiar muchos errores de hardware y de aplicaciones. Los dispositivos y las transferencias de datos pueden fallar de diferentes maneras, ya sea por razones pasajeras, tal como la sobrecarga de la red, o por razones permanentes, tal como un controlador de disco tiene un defecto. Los sistemas operativos a menudo pueden compensar efectivamente las fallas pasajeras. Por ejemplo, una fallo de lectura de disco resulta en un nuevo intento de lectura, y un error en el envío de un mensaje en la red resulta en un re envío, en caso de que el protocolo lo especifique. Desafortunadamente, si un componente importante produce una falla importante, el sistema operativo no lo podrá recuperar.
Como una regla general, una llamada al sistema de I/O podrá retornar un bit de información sobre el estado del llamado, especificando si termino en éxito o fallo. En el sistema operativo UNIX, una variable entera adicional llamada *errno* es usada para retornar un código de error (uno de 100 valores) indicando la naturaleza de la falla (por ejemplo, argumentos fuera de rango, archivo no abierto, etc.). En contraposición, algunos hardwares proveen información detallada del error, aunque los actuales sistemas operativos no son diseñados para comunicar esta información a la aplicación.

Estructura de datos del kernel: el kernel necesita mantener la información de estado sobre el uso de los componentes de I/O. Esto lo hace por medio de una variedad de estructuras de datos que están dentro del kernel, tal como la estructura de la tabla de archivos abiertos. El kernel usa muchas estructuras similares para llevar pista de las conexiones de red, y otras actividades de I/O.
UNIX provee acceso al sistema de archivos para una variedad de entidades, tal como archivos de usuario y los espacios de direcciones de los procesos. Aunque cada una de estas entidades soportan la operación read, la semántica difiere. Por ejemplo, para leer un archivo de usuario, el kernel necesita examinar el buffer de la cache antes de decidir si realiza una I/O de disco. Para copiar la imagen de un proceso, solo es necesario copiar el dato desde la memoria. UNIX encapsula estas diferencias en una estructura uniforme usando la técnica orientada a objetos.
Otros sistemas operativos usan la técnica orientada a objetos más especializada. Por ejemplo, Windows NT usa la implementación de pasaje de parámetros para la I/O. Un pedido de I/O es convertido en un mensaje que es enviado a través del kernel al administrador de I/O y luego al driver del dispositivo, cada uno de los cuales puede cambiar el contenido del mensaje. Para la salida, el mensaje contiene el dato a ser escrito. Para la entrada, el mensaje contiene un buffer para almacenar el dato. Esta implementación puede agregar overhead en comparación con la técnica procedural que usa estructuras de datos compartidas, pero la anterior implementación simplifica la estructura y diseño del sistema de I/O, y agrega flexibilidad.
En resumen, el subsistema de I/O coordina una extensa colección de servicios, el cual están disponibles para las aplicaciones y para otras partes del kernel. El subsistema de I/O supervisa:

- La administración del espacio designado para los archivos y los dispositivos.
- Control de acceso a los archivos y dispositivos.
- Control de la operación (por ejemplo, un modem no puede **seek**).
- Asignar espacio al sistema de archivos.
- Asignar los dispositivos.
- Buffering, caching y spooling.
- Scheduling de I/O.
- Monitoreo del estado de los dispositivos, manejo de errores y recuperación de fallas.
- Configuración de los driver de los dispositivos e inicialización.

Transformación de pedidos de I/O a operaciones de hardware

En esta parte veremos como el sistema operativo conecta un pedido de una aplicación a un conjunto de alambres de red, o a un sector especifico del disco. Consideremos el ejemplo de leer un archivo del disco.

La aplicación se refiere a un dato por medio del nombre del archivo. En un disco, el trabajo del sistema de archivos es, a partir del nombre, buscar en la estructura de directorio para encontrar la ubicación que tiene el archivo. Por ejemplo, en MS-DOS, el nombre se mapea con un número que indica una entrada en la tabla de acceso a archivos, y la entrada de la tabla nos dice en cuales bloques de disco esta alojado el archivo. En UNIX, el nombre se mapea con un número de inode, y el correspondiente inode contiene la información de la ubicación.
Veamos como es la conexión desde el nombre del archivo al controlador del disco. Primero consideremos MS-DOS, un sistema operativo relativamente fácil (supongamos tener el nombre c:\diego\ejecut.exe). La primer parte del nombre de un archivo en MS-DOS, precediendo los dos puntos, es un string que identifica un dispositivo de hardware especifico. Por ejemplo, *c:* es la primer parte de cada nombre de archivo en el disco duro primario. El hecho de que *c:* represente el disco duro primario está establecido en el sistema operativo; *c:* se mapea con una dirección de puerto especifica a través de una tabla de dispositivo. A causa de los dos puntos separadores, el espacio del nombre del dispositivo esta claramente separado del espacio del nombre del sistema de archivos en cada dispositivo. Esta separación hace fácil al sistema operativo asociar funcionalidad extra según cada dispositivo. Por ejemplo, es fácil invocar spooling sobre cualquier archivo que vaya a ser escrito en la impresora.
En cambio, si el espacio del nombre del dispositivo esta incorporado en el espacio del nombre del sistema de archivos, como lo esta en UNIX, los servicios normales son proveídos automáticamente. Si el sistema de archivos provee propiedad y control de acceso a todos los nombre de archivos, entonces los dispositivos tienen los servicios de propiedad y de control de acceso.
UNIX representa los nombres de los dispositivos en el espacio del nombre del sistema de archivos. A diferencia del nombre de un archivo en MS-DOS, el cual tiene los dos puntos separadores, un nombre de path en UNIX no tiene una separación clara de la porción del dispositivo. De hecho, ninguna parte del path es el nombre del dispositivo. UNIX tiene una tabla armada (mount table) que asocia los prefijos de los nombres del path con nombres de dispositivos específicos. Para resolver un nombre del path, UNIX busca el nombre en la tabla para encontrar el prefijo más largo que haga matching con el path. Dicha entrada de la tabla da el nombre del dispositivo. Este nombre de dispositivo tiene también la forma de un nombre en el espacio del nombre del sistema de archivos. Cuando UNIX busca este nombre en la estructura de directorio del sistema de archivos, en lugar de encontrar un número de inode, UNIX encuentra un número de dispositivo <*mayor*, *menor*>. El número de dispositivo *mayor* identifica un driver de dispositivo que deberá ser llamado para manejar la I/O de este dispositivo. El número de dispositivo *menor* es pasado al driver del dispositivo como índice en una tabla de dispositivo. La correspondiente entrada en la tabla de dispositivo da la dirección del puerto o la dirección de memoria mapeada del controlador del dispositivo.
A continuación se describirá el ciclo típico de un pedido de lectura bloqueante. La figura 12.10 sugiere que una operación de I/O requiere muchas etapas, el cual todos juntos consumen gran cantidad de ciclos de CPU.

1. Un proceso invoca una llamada al sistema *read* bloqueante para un archivo que ha sido abierto previamente.
2. El código de la llamada al sistema en el kernel chequea los parámetros para ver si son correctos. En el caso de una entrada, si el dato ya esta disponible en el buffer de la cache, el dato es retornado al proceso y el pedido de I/O es completado.
3. De otra manera, se debe realizar una I/O física por lo que el proceso es eliminado de la cola de listos y es ubicado en la cola de espera para el dispositivo y el pedido de I/O es planificado (scheduler). Cuando es elegido, el subsistema de I/O envía el pedido al driver del dispositivo. Dependiendo del sistema operativo, el pedido es enviado vía un llamado a subrutina o vía un mensaje en el kernel.
4. El driver del dispositivo asigna espacio de buffer en el kernel para recibir el dato, y planifica la I/O. Cuando es elegido, el driver envía comandos al controlador del dispositivo por medio de la escritura en los registros de control del dispositivo.
5. El controlador del dispositivo opera el hardware del dispositivo para realizar la transferencia del dato.

6. El driver puede consultar por estado y datos, o puede tener establecida una transferencia de DMA en la memoria del kernel. Se asume que la transferencia esta administrada por el controlador de DMA (es decir, se opta por la segunda posibilidad), por lo que se generara una interrupción cuando la transferencia se haya completado.
7. El manejador de interrupciones recibe la interrupción vía la tabla vector de interrupciones, almacena cualquier dato necesario, le indica al driver del dispositivo, y retorna de la interrupción.
8. El driver del dispositivo recibe la señal, determina cual pedido de I/O se completo, determina el estado del pedido, y le indica al subsistema de I/O del kernel que el pedido se ha completado.
9. El kernel transfiere el dato o retorna el código al espacio de direcciones del proceso que pidió, y mueve el proceso desde la cola de espera a la cola de listos.
10. Mover el proceso a la cola de listos desbloquea el proceso. Cuando el scheduler asigna la CPU al proceso, el proceso continúa su ejecución.

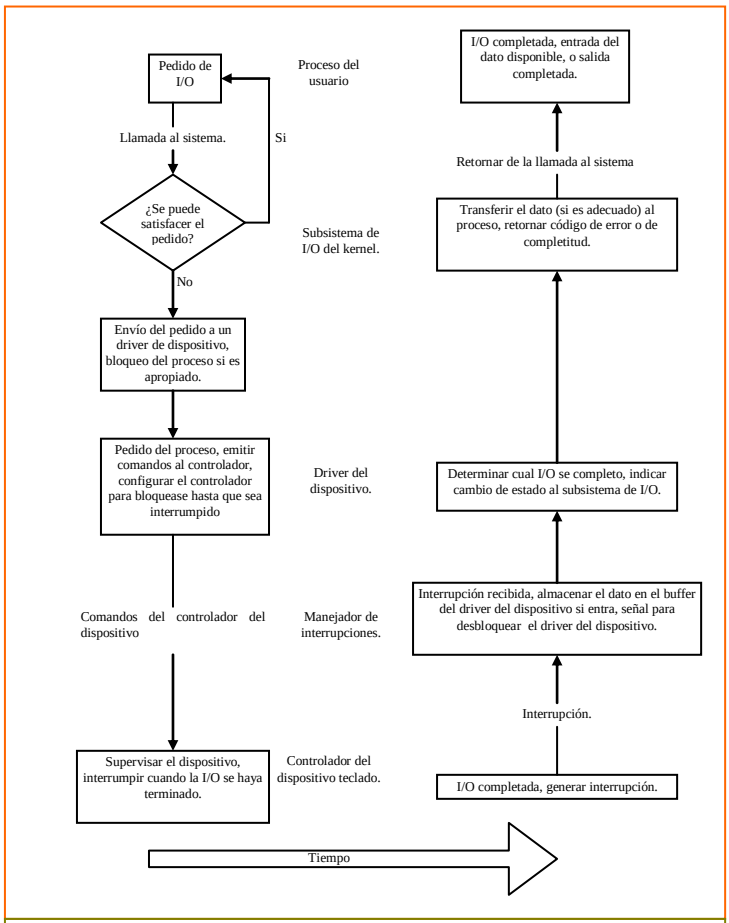


Figura 12.10 Ciclo de vida de un pedido de I/O.

13. Estructura del almacenamiento secundario

El sistema de archivos puede ser visto lógicamente como compuesto de tres partes. Se vio la interfase del usuario y los programadores con el sistema de archivos. Luego se describió las estructuras de datos internas y los algoritmos usados por el sistema operativo para implementar esta interfase. En esta parte se vera el nivel más bajo del sistema de archivos: la estructura de almacenamiento secundario. Primero se describirá los algoritmos de scheduling que planifica el orden de las I/O para mejorar la performance. Luego, se discutirá el formato del disco, y la administración de los bloques booteables, bloques dañados, y espacio de cambio.

Estructura del disco

Los discos proveen el lugar de almacenamiento secundario para las modernas computadoras. Las cintas magnéticas fueron usadas como un medio antiguo de almacenamiento secundario, pero el tiempo de acceso es mucho más lento que el de los discos. Así, las cintas son actualmente usadas principalmente para backup, para el almacenamiento de información infrecuentemente usada, como un medio de transferencia de información entre un sistema y otro, y para almacenar cantidades de datos que son demasiado grandes para discos.

Los drives de discos modernos son direccionados como largos arreglos de una dimensión de bloques lógicos, donde el block lógico es la unidad más chica que se transfiere. El tamaño de un block lógico es usualmente 512 bytes, aunque algunos discos pueden ser formateados a un bajo nivel eligiendo diferentes tamaños de bloques, tal como 1024 bytes.

El arreglo de una dimensión de bloques lógicos es mapeado sobre los sectores del disco secuencialmente. El sector 0 es el primer sector de la primer pista del cilindro más exterior. El mapeo se produce entonces a través de la pista, luego a través del resto de las pistas en el cilindro, y luego a través del resto de los cilindros desde el más exterior hasta el más interior.

Usando éste mapeo, será posible convertir un número de bloque lógico en una dirección de disco que consiste de un número de cilindro, un número de pista en el cilindro, y un número de sector en la pista. En la práctica, es difícil realizar esta traslación por dos razones. Primero, la mayoría de los discos tiene algunos sectores defectuosos, pero el mapeo oculta estos sustituyendo sectores suplentes en otra parte del disco. Segundo, el número de sectores por pista no es constante. La distancia de una pista es desde el centro del disco, aumentando su largo, por lo que puede poseer más sectores. Así, los discos modernos están organizados en zonas de cilindros. El número de sectores por pista es constante en cada zona. Pero a medida que nos movemos desde las zonas interiores hacia las exteriores, el número de sectores por pista aumenta. Las pistas en la zona más exterior típicamente poseen un 40% más de sectores que en las pistas de la zona más interna.

El número de sectores por pista se ha ido incrementando a medida que aumentaba la tecnología de los discos, por lo que es común tener más de 100 sectores por pista en la zona más exterior. Similarmente, el número de cilindros por disco se ha ido incrementando y varios cientos de cilindros es usual.

Scheduling de disco

Una de las responsabilidades del sistema operativo es usar el hardware eficientemente. Para los drives de disco, esto significa tener un tiempo de acceso rápido y un gran ancho de banda de disco. El tiempo de acceso tiene dos componentes principales. El tiempo de búsqueda es el tiempo para que el brazo del disco mueva el cabezal al cilindro conteniendo el sector deseado. La latencia rotacional es el tiempo adicional gastado para que el disco rote al sector deseado. El ancho de banda del disco es el número total de bytes trasferidos, dividido por el tiempo total entre el primer pedido de servicio y la completitud de la última transferencia. Podemos mejorar tanto el tiempo de acceso y el ancho de banda por medio de la planificación de los pedidos de I/O en un buen orden.

Como se vio, cuando un proceso necesita I/O desde o hacia el disco, éste emite una llamada al sistema al sistema operativo. El pedido especifica varias piezas de información:

- Si la operación es de entrada o de salida.
- Cual es la dirección de disco para la transferencia.
- Cual es la dirección de memoria para la transferencia.
- Cual es el número de bytes a ser transferidos.

Si el drive del disco deseado y el controlador están disponibles, el pedido se sirve inmediatamente. En caso de que alguno de ellos este ocupado, cualquier pedido de servicio será ubicado en la cola de pedidos pendientes para este dispositivo. Para un sistema multiprogramado con muchos procesos, la cola del disco podrá tener varios pedidos pendientes. Así, al completarse un pedido, el sistema operativo tiene la posibilidad de elegir cual será el siguiente pedido a servir.

Scheduling FCFS: La forma más simple de planificar el disco es el *first come first served* (FCFS). Este algoritmo es el más simple pero no es el que provee el servicio más rápido. Consideremos un ejemplo, una cola de disco con pedidos de I/O en el orden:

98, 183, 37, 122, 14, 124, 65, 67.

Si la cabeza del disco esta inicialmente en el cilindro 53, primero se moverá desde el cilindro 53 al 98, luego al 183, 37, 122, 14, 124, 65, y finalmente al 67, para un total de 640 cilindros del movimiento del cabezal (Figura 13.1).

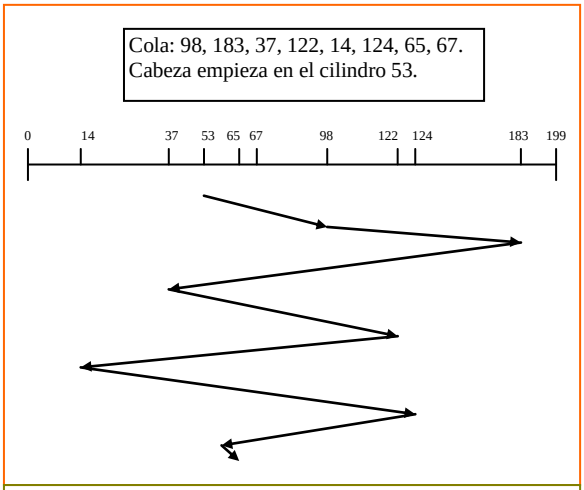


Figura 13.1 Scheduling de disco FCFS.

Scheduling SSTF: Se vería razonable servir todos los pedidos cercanos a la actual posición del cabezal, antes de haberla movido lejos para servir otros pedidos. Esta es la idea básica del algoritmo shortest sep time first (SSTF). Este algoritmo selecciona el pedido con el mínimo tiempo de búsqueda desde la actual posición del cabezal. Ya que el tiempo de búsqueda se incrementa con el número de cilindros atravesados por el cabezal, SSTF elige el pedido pendiente más cercano a la actual posición del cabezal.

Para el ejemplo anterior, y comenzando desde la posición 53 la cabeza va al pedidos en la posición 65. Una vez que esta en el cilindro 65, el siguiente pedido más cercano es el que esta en la posición 67. Desde ahí, el pedido en la posición 37 es el que esta más cerca. Luego se sirve el 14, el 98, el 122, el 124 y por último el 183. Este método provoco un total de movimiento de 236 cilindros (Figura 13.2).

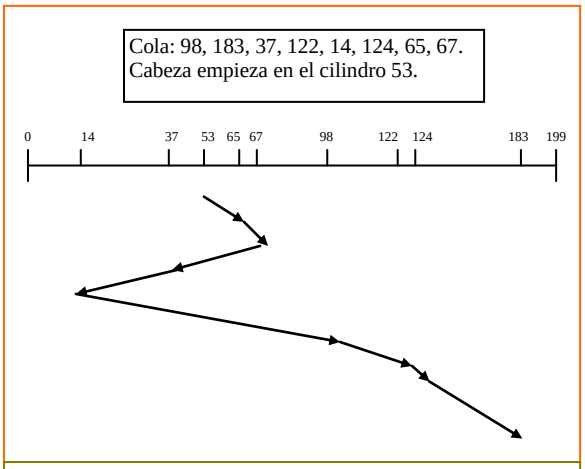


Figura 13.2 Scheduling de disco SSTF.

Este algoritmo puede causar inanición de pedidos. Recuerde que los pedidos pueden arribar en cualquier momento. Supongamos que tenemos dos pedidos en la cola, para los cilindros 14 y 186, y luego de servir el pedido en el cilindro 14 llega otro pedido cercano al 14. Así, este pedido nuevo será servido primero, haciendo que el 186 espere. Mientras este pedido cercano al 14 es servido, puede llegar otro cercano a éste. En teoría, un continuo flujo de pedidos cercanos al 14 podrían llegar, provocando que el pedido del cilindro 186 espere indefinidamente.
Este algoritmo no es el óptimo. En el ejemplo, hubiera sido mejor mover la cabeza desde la 53 a la 37, aun aunque el 14 este más cerca que el 37. Este cambio provocara que el número de movimientos de la cabeza sea de 208 (53, 37, 14, 65, 67, 98, 122, 124, 183).

Scheduling SCAN: En el algoritmo SCAN, el brazo del disco comienza en un lado del disco, y se va moviendo hacia el otro lado, sirviendo los pedidos que va encontrando en su camino, hasta que llega al otro lado del cilindro, en donde invierte su dirección, sirviendo todos los pedidos que encuentre en esta nueva dirección.
Antes de aplicar este algoritmo a nuestro ejemplo, se debe conocer en que dirección ira el cabezal desde la posición 53. Si el brazo del disco va hacia la dirección 0, servirá los pedidos 37 y 14. En el cilindro 0, el brazo comenzara a ir hacia el otro lado, sirviendo los pedidos 65, 67, 98, 122, 124, y 183 (Figura 13.3). Si un pedido arriba en la cola justo enfrente del cabezal, será servido inmediatamente; un pedido que llega justo atrás del cabezal tendrá que esperar hasta que el brazo se mueva hacia la otra dirección.

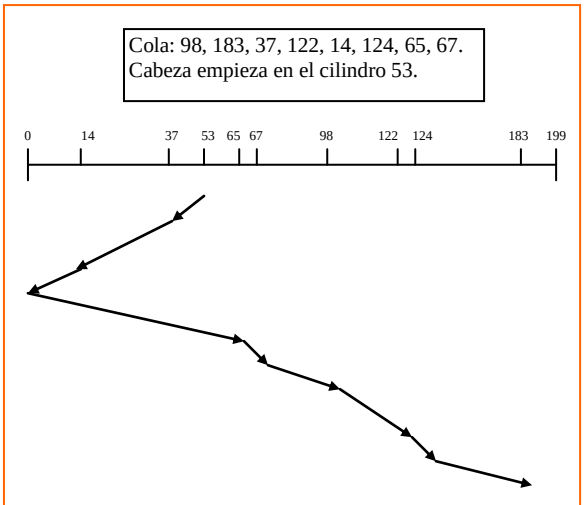


Figura 13.3 Scheduling de disco SCAN.

Scheduling C-SCAN: Circular SCAN es una variación de SCAN el cual esta diseñada para proveer una espera de tiempo más uniforme. Como SCAN, C-SCAN mueve la cabeza de un lado a otro del disco, sirviendo los pedidos que encuentra en su camino. Cuando la cabeza encuentra el fin del disco, retorna inmediatamente al inicio del disco, sin servir ningún pedido que encuentre en este retorno. El scheduling C-SCAN trata a los cilindros como una lista circular (Figura 13.4).

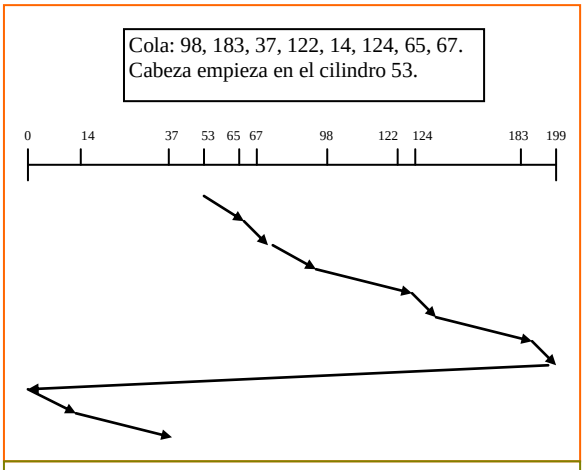


Figura 13.4 Scheduling de disco C-SCAN.

Scheduling LOOK: Note que ambos algoritmos, SCAN y C-SCAN mueven el brazo del disco desde un lado hacia el otra e invierten la dirección cuando llegan hasta el final del disco. En la practica ninguno de estos algoritmos es implementado ya que no es necesario ir hasta el final del disco si se sabe que no existirá ningún pedido. Es más común que el brazo vaya hasta donde esta el final del pedido en cada dirección. Así, las versiones de SCAN y C-SCAN son llamadas LOOK y C-LOOK (Figura 13.5).

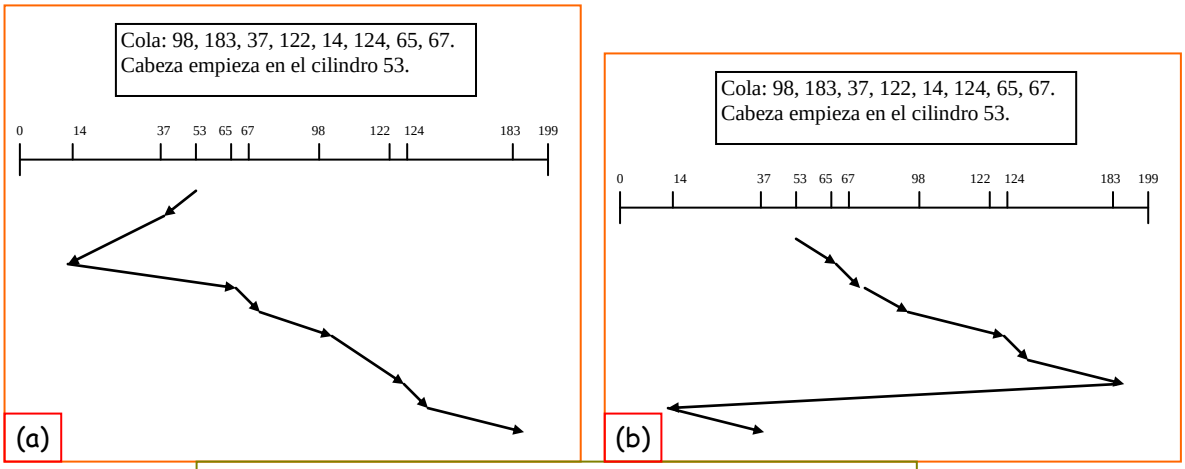


Figura 13.5 Scheduling de disco (a) LOOK (b) C-LOOK.

Administración del disco

Veamos otros aspectos de los cuales el sistema operativo es responsable.

Formateo de disco: un nuevo disco magnético es una pizarra blanca. Antes de que un disco pueda almacenar datos, se debe dividir en sectores que el controlador del disco pueda leer o escribir. Esto es llamado formateo a bajo nivel, o formateo físico. Este tipo de formateo llena el disco con una estructura de datos especial para cada sector. La estructura de datos para un sector típicamente consiste de una cabecera, un área de datos (usualmente de un tamaño de 512 bytes), y un trailer. La cabeza y el trailer contienen información usada por el controlador del disco, tal como el número de sector y código corrector de errores (ECC). Cuando el controlador escribe un sector de datos durante una I/O normal, el ECC es

modificado con un valor calculado con todos los bytes del área de datos. Cuando se lee el sector, el ECC es recalculado y se compara con el valor almacenado. Si el valor almacenado y el calculado son diferentes, la desigualdad indica que el área de datos del sector esta adulterada y el sector del disco puede estar malo. El ECC es un código corrector de errores ya que contiene bastante información que si solo 1 o 2 bits han sido adulterados, el controlador puede identificar cuales bits han cambiado, y puede calcular cual es su valor correcto. El procedimiento del ECC es realizado automáticamente por el controlador cuando se lee o escribe un sector.
La mayoría de los discos duros son formateados en bajo nivel en la fabrica como parte del proceso de fabricado. Esta información permite al fabricante testear el disco.
Para usar un disco para mantener archivos, el sistema operativo necesita aun registrar su propia estructura de datos en el disco. Esto lo hace en dos pasos. El primer paso es particionar el disco en uno o más grupos de cilindros. El sistema operativo puede tratar a cada partición como un disco separado. Por ejemplo, una partición puede contener una copia del código ejecutable del sistema operativo, mientras otra mantiene los procesos del usuario. Luego de la partición, el segundo paso es llamado formateo lógico, o "hacer el sistema de archivos". En este paso, el sistema operativo almacena la estructura de datos del sistema de archivos inicial en el disco.
Algunos sistemas operativos dan a programas especiales la habilidad de usar una partición del disco como un gran arreglo secuencial de bloques lógicos, sin ninguna estructura de datos del sistema de archivos. Este arreglo es a veces llamado raw I/O. Por ejemplo, algunos sistemas de base de datos prefieren raw I/O ya que esto les permite el control de la ubicación exacta del disco donde cada registro de la base de datos esta almacenado. Raw I/O desvía a todos los servicios del sistema de archivos, tal como buffer cache, prefetching, asignación de espacio, nombre de archivos, y directorios. Podemos hacer algunas aplicaciones más eficientes implementando sus propios servicios de almacenamiento de propósito general sobre una partición *raw*, pero la mayoría de las aplicaciones se comportan mejor cuando utilizan los servicios regulares del sistema de archivos.

Bloque de booteo: para que una computadora empiece a correr (por ejemplo, cuando es prendida o reiniciada), necesita tener un programa inicial para correr (bootstrap). Este programa tiende a ser simple: inicializa todos los aspectos del sistema, desde los registros de la CPU a los controladores de los dispositivos y los contenidos de la memoria principal, y luego arranca el sistema operativo. Para hacer este trabajo, éste encuentra el kernel (núcleo) del sistema operativo en disco, carga este kernel en memoria, y salta a la dirección inicial para comenzar la ejecución del sistema operativo.
En la mayoría de las computadoras, el bootstrap es almacenado en memoria de solo lectura (ROM). Esta ubicación es conveniente, ya que la ROM no necesita ser inicializada y esta en un lugar fijo que el procesador puede comenzar a ejecutar cuando se da energía o se resetea. Y, ya que la ROM es de solo lectura, no puede ser infectada por ningún virus. El problema es que cambiar el código de este bootstrap requiere cambiar los chips de hardware de la ROM. Por esta razón, la mayoría de los sistemas almacenan un diminuto programa *bootstrap loader* en la ROM de booteo, cuyo único trabajo es traer un programa completo de bootstrap de disco. El programa completo de bootstrap puede ser cambiado fácilmente: una nueva versión simplemente se escribe en disco. El programa completo bootstrap es almacenado en una partición llamada los bloques boot, en un lugar fijo del disco. Un disco que tiene una partición boot es llamado un disco boot o disco del sistema.
El código en la ROM boot instruye al controlador de disco para leer los bloques boot en memoria (los drivers de los dispositivos no están cargados en este punto), y luego comienza la ejecución del código. El programa completo bootstrap es más sofisticado que el cargador bootstrap en la ROM boot, y es capaz de cargar el sistema operativo entero, desde una ubicación no fija de disco, y comenzar a correr el sistema operativo. Aun así, el código completo de bootstrap puede ser muy chico. Por ejemplo, el sistema operativo MS-DOS usa un bloque de 512 bytes para su programa boot (figura 13.6).

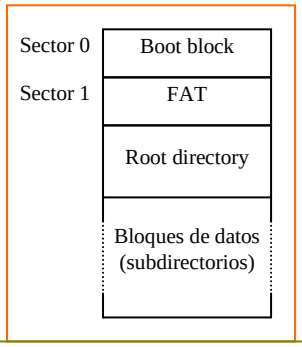


Figura 13.6 Esquema de un disco en MS-DOS.

Bloques malos: Ya que el disco tiene partes que se mueven y pequeñas tolerancias (recordamos que la cabeza del disco se mueve a través de la superficie del disco), ellos están propensos a fallas. Algunas veces la falla es completa, y el disco se debe reemplazar, y su contenido restaurados desde una copia de seguridad. Es muy frecuente que uno o dos sectores se rompan. Algunos discos pueden venir desde fabrica con bloques malos. Dependiendo del disco y controlador en uso, éstos bloques se manejan de una forma muy variada.

En un disco simple, tal como un disco con controladores IDE, los bloques malos son manejados manualmente. Por ejemplo, el comando *format* en MS-DOS hace un formateo lógico y, parte de este proceso, examina el disco para encontrar bloques malos. Si format encuentra un bloque malo, escribe un valor especial en la correspondiente entrada de la FAT para avisar a las rutinas que ese bloque no debe ser usado. Si los bloques se rompen durante una operación normal, un programa especial (tal como *chkdsk*) se debe correr manualmente para buscar los bloques rotos y para bloquearlos. Los datos que residen en los bloques rotos normalmente se pierden.

Los discos más sofisticados, tal como los discos SCSI usados en PCs, son inteligentes sobre la recuperación de bloques malos. El controlador mantiene una lista de bloques malos en el disco. La lista es inicializada durante el formateo en bajo nivel en la fabrica, y es modificada según aparezcan nuevos bloques. El formateo en bajo nivel también setea los sectores sobrantes no visibles para el sistema operativo. El controlador puede ser avisado de cambiar un bloque que esta averiado por uno sobrante. Este esquema es conocido como *forwarding*:

- El sistema operativo trata de leer el bloque lógico 87.
- El controlador calcula el ECC y encuentra que el sector es malo. Este reporta esto al sistema operativo.
- La siguiente vez que el sistema es reiniciado, se correrá un comando especial para decirle al controlador SCSI que reemplace el bloque averiado por uno sobrante.
- Luego de esto, cuando el sistema haga un pedido al bloque 87, el pedido será trasladado en la dirección del bloque reemplazado por el viejo por el controlador.

Administración del espacio de swap

Administrar el espacio de cambio es otra tarea de bajo nivel del sistema operativo. La memoria virtual usa espacio de disco como una extensión de la memoria principal. Ya que los accesos a disco son mucho más lento que los accesos a memoria, usar espacio de cambio tiene un gran efecto en la performance del sistema. El mayor objetivo para el diseño e implementación del espacio de cambio es proveer el mejor throughput para el sistema de memoria virtual. En esta parte, se vera como se usa este espacio, donde se ubica en disco, y como se administra.

Uso del espacio de swap (swap-space): El espacio de cambio se usa de varias maneras por diferentes sistemas operativos, dependiendo de los algoritmos de administración de memoria implementados. Por ejemplo, los sistemas que implementan swaping pueden usar espacio de cambio para mantener la imagen

completa de procesos, incluyendo los segmentos de código y de datos. Los sistemas de paginado pueden simplemente almacenar las páginas que se han sacado de la memoria principal. La cantidad de espacio de cambio necesario puede, por lo tanto, variar dependiendo de la cantidad de memoria física, la cantidad de memoria virtual, y de la forma en el cual se utilice la memoria virtual. Este rango puede ir desde unos pocos megabytes hasta cientos de megabytes o más de espacio de disco.
Algunos sistemas operativos, tal como UNIX, permite el uso de múltiples espacios de swap. Estos espacios son usualmente puestos en discos separados.

Ubicación del espacio de swap: existen muchos lugares donde el espacio de swap puede residir: puede ocupar un lugar fuera del espacio del sistema de archivos normal, o puede estar en una partición de disco separada. Si el espacio de cambio es simplemente un gran archivo en el sistema de archivo, las normales rutinas del sistema de archivo pueden ser usadas para crearlo, nombrarlo, y asignarle espacio. Esta idea es por lo tanto fácil de implementar. Desafortunadamente, esto es ineficiente. Navegar por la estructura de directorio y las estructuras de datos para la asignación de lugar en el disco toma tiempo, y accesos a disco extras. La fragmentación externa se puede incrementar mucho. Podemos mejorar la performance guardando la información de la ubicación del bloque en memoria física, y usar herramientas especiales para asignar bloques contiguos para el cambio de archivo (swap file), pero el costo de navegar por el sistema de archivos se mantiene.
Es más común que el espacio de cambio sea creado en una partición diferente. Ni el sistema de archivos ni el directorio se ubica en este lugar. En vez de esto, se usa un administrador de almacenamiento de espacio de cambio para asignar y desasignar los bloques. Este administrador usa algoritmos que son óptimos en cuanto a velocidad, en lugar de ser eficientes en el espacio de almacenamiento. La fragmentación interna se puede incrementar, pero este tradeoff es aceptable ya que los datos en el espacio de cambio generalmente viven mucho menos tiempo que los archivos en el sistema de archivos, y el área de cambio puede ser accedida muchas más veces. Pero esta idea crea un espacio de cambio cuando el disco es particionado. Agregar más espacio de cambio al ya existente solo se puede hacer vía una repartición del disco o vía la agregación de otro espacio de cambio en otra parte.

Administrador del espacio de swap: Para ilustrar el método utilizado para administrar el espacio de búsqueda, veremos la evolución del swapping y paginado de UNIX. UNIX comenzó con una implementación de swapping que copiaba procesos enteros entre contiguas regiones de disco y memoria. UNIX evoluciono a una combinación de swapping y paginado cuando estuvo disponible el hardware de paginado.
En 4.3BSD, el espacio de cambio se asignaba a un proceso cuando se comenzaba el proceso. Se seteaba bastante espacio para mantener el programa, conocido como el segmento de texto, y el segmento de datos del proceso. Cuando el proceso comenzaba, su texto era paginado desde el sistema de archivo a memoria. Cuando se necesitaba, estas páginas eran sacadas de memoria y llevadas al espacio de cambio, y cuando se volvían a necesitar se traían nuevamente a memoria, por lo que el sistema de archivos solo era consultado una vez para cada página de texto. Las páginas del segmento de datos eran traídas también desde el sistema de archivo, o eran creadas, y eran sacadas (ubicadas en el espacio de cambio) o traídas nuevamente a memoria según se necesitaba. Una optimización (por ejemplo, cuando dos usuarios corrían el mismo editor) es que los procesos con idénticas páginas de texto, compartían esas páginas, tanto en el espacio de memoria física como en el espacio de cambio.
Dos regiones de cambio por proceso son usadas por el kernel para llevar pista del uso del espacio de cambio. El segmento de texto es de tamaño fijo, por lo que su espacio de cambio es asignado en pedazos de 512 bytes, excepto por el último pedazo, el cual contiene el resto de las páginas, incrementándose este último pedazo en incrementos de 1K (Figura 13.7).

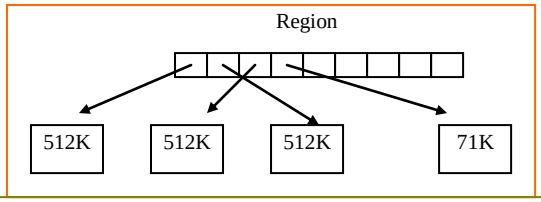


Figura 13.7 Región de cambio del segmento de texto.

La región en el cambio del segmento de dato es más complicado, ya que el segmento de dato puede crecer con el tiempo. La región es de tamaño fijo, pero contiene direcciones para bloques de tamaño variante. Dando un índice i, un bloque apuntaba a una entrada de una región, de tamaño 2^i * 16K, para un máximo de 2 megabytes (figura 13.8). El tamaño mínimo y máximo de bloque es variable y puede ser cambiado. Cuando un proceso trata de aumentar su segmento de datos más allá del final del bloque asignado en su área de cambio, el sistema operativo asignara otro bloque, el doble de grande que el anterior.

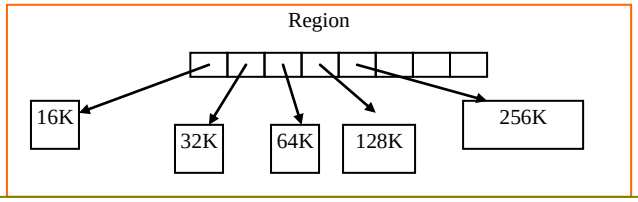


Figura 13.8 Región de cambio del segmento de datos.