UNIVERSIDAD ESTATAL A DISTANCIA
Escuela de Ciencias Exactas y Naturales
Carrera: Ingeniería Informática

Enrique Gómez Jiménez

Material complementario para el curso Sistemas Operativos

Código 881

2009

Edición Académica:
Ana Láscaris-Comneno Slepuhin

Encargada de Cátedra y especialista de contenidos:
Karol Castro Chávez

Revisión filológica:
Óscar Alvarado Vega

Tabla de contenidos

Capítulo 1: Introducción

1.1. ¿Qué es un sistema operativo?

Un **sistema operativo** (SO) es un conjunto de rutinas o extensiones de *software* que permiten la administración y gestión del *hardware*. Consiste básicamente en rutinas de control que hacen funcionar una computadora y proporcionan un entorno para la ejecución de los programas. Permite, además, la interfaz adecuada para que el usuario se comunique con el *hardware* del computador.

El sistema operativo debe cumplir cuatro tareas fundamentales en un computador que se encuentre encendido:

1. Proporcionar la interfaz gráfica al usuario para que el usuario se comunique con el equipo.
2. Administrar los dispositivos de *hardware* del equipo.
3. Administrar y mantener los sistemas de archivos en los dispositivos de almacenaje del equipo.
4. Apoyar a otros programas instalados en el equipo proveyéndole de servicios adecuados.

Figura 1: Tareas fundamentales de un sistema operativo.
Fuente: Gómez Jiménez, Enrique.

1.1.1. Sistema operativo como máquina extendida

En esta perspectiva, la función del sistema operativo es la de presentar al usuario el equivalente de una máquina extendida o máquina virtual que sea más fácil de programar que el *hardware* subyacente. Dicho en otras palabras, la máquina virtual es más fácil de programar que la máquina pura. Además, para una misma familia de máquinas, aunque tengan componentes diferentes (por ejemplo, monitores de distinta resolución o discos duros de diversos fabricantes), la máquina virtual puede ser idéntica: el programador ve exactamente la misma interfaz.

1.1.2. Sistema operativo como administrador de recursos

Desde este punto de vista, la labor del sistema operativo es la de proporcionar una asignación ordenada y controlada de los procesadores, memorias y dispositivos de entrada y salida para varios programas que compiten entre ellos.

En resumen, el sistema operativo debe:

- Llevar la cuenta acerca de quién está usando qué recursos.
- Otorgar recursos a quienes los solicitan (siempre que el solicitante tenga derechos adecuados sobre el recurso).
- Arbitrar en caso de solicitudes conflictivas.

1.2. Historia de los sistemas operativos

1.2.1. Primera generación (1945 – 1955): bulbos y conexiones

En esta década aparecen los sistemas de procesamiento por lotes, donde los trabajos se reunían por grupos o lotes. Cuando se ejecutaba alguna tarea, esta tenía control total de la máquina. Al terminar cada tarea, el control era devuelto al sistema operativo, el cual limpiaba, leía e iniciaba la siguiente tarea. Aparece el concepto de nombres de archivo del sistema para lograr independencia de información. Los laboratorios de investigación de General Motors poseen el crédito de haber sido los primeros en poner en operación un sistema operativo para su IBM 701.

1.2.2. Segunda generación (1955 – 1965): transistores

En esta generación se desarrollan los sistemas compartidos con multiprogramación, en los cuales se utilizan varios procesadores en un solo sistema, con la finalidad de incrementar el poder de procesamiento de la máquina. El programa especificaba tan solo que un archivo iba a ser escrito en una unidad de cinta con cierto número de pistas y cierta densidad. El sistema operativo localizaba entonces una unidad de cinta disponible con las características deseadas, y le indicaba al operador que montara una cinta en esa unidad.

1.2.3. Tercera generación (1965 – 1980): circuitos integrados

En esta época surge la familia de computadores IBM/360 diseñados como sistemas para uso general, por lo que requerían manejar grandes volúmenes de información de distinto tipo. Esto provocó una nueva evolución de los sistemas operativos: los sistemas de modos **múltiples**, que soportan simultáneamente procesos por lotes, tiempo compartido, procesamiento en tiempo real y multiprocesamiento.

1.2.4. Cuarta generación (1980 – 1990): computadoras personales

Los sistemas operativos conocidos en la época actual son los considerados **sistemas de cuarta generación**. Con la ampliación del uso de redes de computadoras y del procesamiento en línea, es posible obtener acceso a computadoras alejadas geográficamente, a través de varios tipos de terminales. Con estos sistemas operativos aparece el concepto de **máquinas virtuales**, en el cual el usuario no se involucra con el *hardware* de la computadora con la que se quiere conectar, y en su lugar el usuario observa una interfaz gráfica creada por el sistema operativo.

1.2.5. La ontogenia recapitula la filogenia

La evolución del *hardware* pareciera un proceso análogo al argumentado por Ernst Haeckel sobre la evolución biológica: desde el *mainframe* hasta el computador de bolsillo, su arquitectura ha pasado por diversas etapas. Del mismo modo ha pasado con la evolución de los sistemas operativos.

1.3. La variedad de sistemas operativos

Debido a la evolución de los sistemas operativos, fue necesario realizar una clasificación. Considerando las diferencias existentes entre sus componentes, los podemos clasificar en:

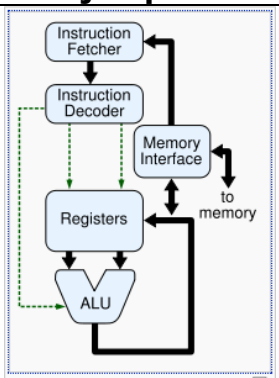
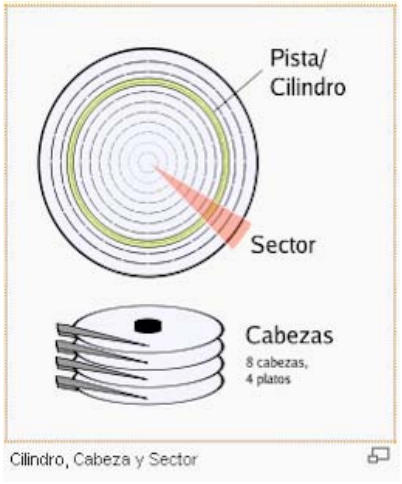
- Sistemas operativos de *mainframes*
- Sistemas operativos de servidor
- Sistemas operativos de multiprocesador
- Sistemas operativos de computadora personal
- Sistemas operativos de tiempo real
- Sistemas operativos integrados
- Sistemas operativos de tarjeta inteligente

Sistema operativo	Características

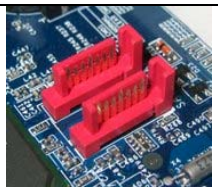
de *mainframes*	• Se orientan al multiprocesamiento.
	  ○ Lotes
	  ○ Transacciones
	  ○ Tiempo compartido
	• Ejemplo: OS/390 de IBM

de servidor	• Tienen múltiples usuarios a nivel de red. • Ejemplo: Linux, Windows
de multiprocesador	• Cuentan con dos o más microprocesadores. • Ejemplo: sistema de red neuronal
de computadora personal	• Interfaz amigable con el usuario • Interfaz monousuario • Ejemplo: Windows XP, Linux
de tiempo real	• Su parámetro es el tiempo. • Ejemplo: Sistema automatizado de control industrial
integrados	• De bolsillo • Ejemplo: Windows CE, PDA
de tarjeta inteligente	• Son empotradas en tarjetas inteligentes. • Ejemplo: empotradas mediante *applets* de Java

1.4. Repaso de *hardware* de cómputo

Componente	Descripción	Ejemplo
CPU (***C**entral **P**rocessing **U**nit*)	• Es el componente en una computadora digital que interpreta las instrucciones y procesa los datos contenidos en los programas de computadora. • La operación fundamental de la mayoría de los CPU, sin importar la forma física que tomen, es ejecutar una secuencia de instrucciones almacenadas llamadas **programa**. • Manejan registros (contador de palabras, apuntadores de pila, buffer temporales de registros, etc.) e instrucciones.	 Diagrama de bloques de un CPU simple
memoria	• Se refiere a componentes de una computadora, dispositivos y medios de grabación que retienen datos informáticos durante algún intervalo de tiempo. • Proporcionan una de las principales funciones de la computación moderna, la retención de información. • Es uno de los componentes fundamentales de todas las computadoras modernas que, acoplados a una Unidad Central de Proceso (CPU por su acrónimo en inglés *Central Processing Unit*), implementa lo fundamental del modelo de computadora de Von Neumann, usado desde los años 1940. • En la actualidad, memoria suele referirse a una forma de almacenamiento de estado sólido conocido como memoria RAM (memoria de acceso aleatorio, RAM por sus siglas en inglés *Random Access Memory*). Otras veces se refiere a otras formas de almacenamiento rápido pero temporal.	  Cilindro, Cabeza y Sector

	• De forma similar, se refiere a formas de almacenamiento masivo como discos ópticos, y tipos de almacenamiento magnético como discos duros y otros tipos de almacenamiento más lentos que las memorias RAM, pero de naturaleza más permanente. Estas distinciones contemporáneas son de ayuda porque son fundamentales para la arquitectura de computadores en general.	
dispositivos de entrada / salida	• Abreviado **E/S** o **I/O** (del original en inglés *input/output*), es la colección de interfaces que usan las distintas unidades funcionales (subsistemas) de un sistema de procesamiento de información para comunicarse unas con otras, o las señales (información) enviadas a través de esas interfaces. • Las entradas son las señales recibidas por la unidad, mientras que las salidas son las señales enviadas por esta. • Los teclados y ratones se los considera dispositivos de entrada de una computadora, mientras que los monitores e impresoras son vistos como dispositivos de salida de una computadora. • Los dispositivos típicos para la comunicación entre computadoras realizan las dos operaciones, tanto entrada como salida, y entre otros se encuentran los módems y tarjetas de red.	
buses	• En arquitectura de computadores, un bus puede conectar lógicamente varios periféricos sobre el mismo conjunto de cables. • Un bus es un conjunto de conductores eléctricos en forma	 Conectores SCSI

	de pistas metálicas impresas sobre la tarjeta madre del computador, por donde circulan las señales que corresponden a los datos binarios del lenguaje máquina con que opera el Microprocesador.
	• Hay tres clases de buses: bus de datos, bus de direcciones y bus de control.
	  ○ El bus de datos mueve los datos entre los dispositivos del *hardware* de entrada como el teclado, el ratón, etc.; de salida como la Impresora, el Monitor; y de almacenamiento como el disco duro, el disquete o la memoria-*flash*.
	  ○ El bus de direcciones está vinculado al bloque de control del CPU para tomar y colocar datos en el sub-sistema de memoria durante la ejecución de los procesos de cómputo. En un instante dado, un conjunto de 32 bits nos sirve para calcular la capacidad de memoria en el CPU.
	  ○ El bus de control transporta señales de estado de las operaciones efectuadas por el CPU. El método utilizado por el ordenador para sincronizar las distintas operaciones es por medio de un reloj interno que posee el ordenador, que facilita la sincronización y evita las colisiones de operaciones (unidad de control).
		
		Puerto SATA
		
		 
		Video AGP

1.5. Conceptos de los sistemas operativos:

Concepto	Descripción
procesos	Es un concepto manejado por el sistema operativo. Consiste en el conjunto formado por: • Las instrucciones de un programa destinadas a ser ejecutadas por el microprocesador. • Su estado de ejecución en un momento dado, esto es, los valores de los registros del CPU para dicho programa. • Su memoria de trabajo, es decir, la memoria que ha reservado y sus contenidos. • Otra información que permite al sistema operativo su planificación.
bloqueos	El bloqueo de un proceso consiste en bloquear uno o más procesos para ejecutar otro(s). Existen bloqueos en los cuales puede indeterminarse la ejecución posterior el proceso bloqueado. En sistemas operativos, el **bloqueo mutuo** (también conocido como interbloqueo, traba mortal, *deadlock*, abrazo mortal) es el bloqueo permanente de un conjunto de procesos o hilos de ejecución en un sistema concurrente que compiten por recursos del sistema, o bien se comunican entre ellos. A diferencia de otros problemas de concurrencia de procesos, no existe una solución general para los interbloqueos.
administración de la memoria	La función de la memoria de ordenador es retener durante cierto período información que necesita el sistema para funcionar correctamente, y los procesos o archivos más empleados en la utilización del ordenador. La multiprogramación de partición fija permite que varios procesos usuarios compitan al mismo tiempo por los recursos del sistema: Un trabajo en espera de E/S cederá el CPU a otro trabajo que esté listo para efectuar cómputos. Existe paralelismo entre el procesamiento y la E/S. Se incrementa la utilización del CPU y la capacidad global de ejecución del sistema. Es necesario que varios trabajos residan a la vez en la memoria principal. En la partición variable no hay límites fijos de memoria, es decir que la partición de un trabajo es su propio tamaño. Se consideran **esquemas de asignación contigua**, dado que un programa debe ocupar posiciones adyacentes de almacenamiento. Los procesos que terminan dejan disponibles espacios de memoria principal llamados **agujeros**.
entrada / salida	Es la colección de interfaces que usan las distintas unidades funcionales (subsistemas) de un sistema de procesamiento de información para comunicarse unas con otras, o las señales (información) enviadas a través de esas interfaces.

	Las entradas son las señales recibidas por la unidad, mientras que las salidas son las señales enviadas por esta.
archivos	Un sistema de archivos consta de tipos de datos abstractos, que son necesarios para el almacenamiento, organización jerárquica, manipulación, navegación, acceso y consulta de datos. La mayoría de los sistemas operativos poseen su propio sistema de archivos. Los sistemas de archivos son representados ya sea textual o gráficamente, utilizando gestores de archivos o *shells*. En modo gráfico, a menudo son utilizadas las metáforas de carpetas (directorios) que contienen documentos, archivos y otras carpetas. Un sistema de archivos es parte integral de un sistema operativo moderno. El *software* del sistema de archivos es responsable de la organización de estos sectores en archivos y directorios, y mantiene un registro de qué sectores pertenecen a qué archivos y cuáles no han sido utilizados. En la realidad, un sistema de archivos no requiere necesariamente de un dispositivo de almacenamiento de datos, sino que puede ser utilizado también para acceder a datos generados dinámicamente, como los recibidos a través de una conexión de red.
seguridad	La seguridad de un sistema operativo se orienta hacia la protección de archivos y del acceso al equipo y a los programas. La seguridad interna está relacionada con los controles incorporados al *hardware* y con el sistema operativo para asegurar los recursos del sistema.
shell	Se encarga de ejecutar las llamadas al sistema. El *shell* es el intérprete de comandos; a pesar de no ser parte del sistema operativo, hace un uso intenso de muchas características del sistema operativo. Por tanto, sirve como un buen ejemplo de la forma en que se pueden utilizar las llamadas al sistema. También es la interfaz primaria entre un usuario situado frente a su terminal y el sistema operativo. Cuando algún usuario entra al sistema, un *shell* se inicia. El *shell* tiene la terminal como entrada y como salida estándar. Este da inicio al teclear solicitud de entrada, carácter en forma de un signo de pesos, el cual indica al usuario que el *shell* está esperando un comando. En MS-DOS normalmente aparece la letra de la unidad, seguida por dos puntos (:), el nombre del directorio en que se encuentra y, por último, el signo de *mayor que* (>).

1.6. Llamadas al sistema

En las **llamadas al sistema** se define la interfaz entre el sistema operativo y los programas de usuario. Las llamadas principales se agrupan en procesos, archivos, directorios y otros. Son las llamadas que ejecutan los programas de aplicación para pedir algún servicio al sistema operativo.

Cada sistema implementa un conjunto propio de llamadas al sistema. Ese conjunto de llamadas es la interfaz del sistema operativo frente a las aplicaciones. Constituyen el lenguaje que deben usar las aplicaciones para comunicarse con el sistema operativo. Las principales llamadas al sistema operativo son:

- Control de procesos: fin, abortar, cargar, ejecutar, crear, finalizar, obtener y establecer atributos, espera, asignar y liberar memoria.

- Manipulación de archivos: crear y eliminar archivo, abrir y cerrar, leer, escribir, reposicionar, obtener y establecer atributos.

- Manipulación de directorios: crear, modificar, eliminar directorios de archivos.

Algunos ejemplos en Java para estos procesos del sistema operativo son:

Para crear un archivo:

```
import java.io.*;
public class CreateFile1{
   public static void main(String[] args) throws IOException{
     File f;
     f=new File("myfile.txt");
     if(!f.exists()){
     f.createNewFile();
     System.out.println("Nuevo Archivo\"myfile.txt\" creado");
     }
   }
}
```

Para leer un archivo:

```
  import java.io.*;
```

```
public class ReadFile{
    public static void main(String[] args){
     try {

// Establece un archivo lector para leer el archivo un carácter cada vez

    FileReader input = new FileReader(args[0]);

// Filtrar a través de un buffer leyendo línea por vez
   BufferedReader bufRead = new BufferedReader(input);

          String line;    // cadena que contiene del archivo la línea actual
         int count = 0;  // contador del numero de línea.

          //Lee la primera línea
          line = bufRead.readLine();
         count++;

    //Lee a través del archive una línea a la vez.  Imprime el # y la línea
        while (line != null){
          System.out.println(count+": "+line);
              line = bufRead.readLine();
         count++;
          }

          bufRead.close();

      }catch (ArrayIndexOutOfBoundsException e){
    // Se genera una excePCión
          System.out.println("Usage: java ReadFile filename\n");

       }catch (IOException e){
          //si se genera otra, imprime  la traza de la pila
          e.printStackTrace();
       }

   }//fin main
```

1.7. Estructura del sistema operativo

Los cinco diseños de un sistema operativo son:

Diseño	Descripción
monolíticos	• Los procesos que se llaman entre sí se construyen como un árbol general. • Los procedimientos son: principal, de servicio y utilitarios.
capas	Se organizan en capas. En cada capa residían procesos que se relacionaban lógicamente entre sí.
máquinas virtuales	Poseen un *software* que se ejecuta en el *hardware* y se realiza en ella la multiprogramación.
*exokernel*s	Con la VM/370 y la virtualización, cada proceso de usuario obtiene una copia exacta de la computadora real. Con el 8086 de Pentium, cada proceso de usuario obtiene una copia exacta de una computadora distinta. El *exokernel* se ubica en la capa más baja de máquina virtual. Su labor es asignar recursos a las máquinas virtuales, y asegurarse de que ninguna utilice los recursos de otra.
cliente servidor	Consiste en implementar casi todo el sistema operativo en procesos de usuario (cliente) quien solicita servicios a un proceso de *kernel* (servidor)

Capítulo 2: Procesos y subprocesos

El **proceso** es manejado por el sistema operativo. Consiste en el conjunto formado por:

- Las instrucciones de un programa destinadas a ser ejecutadas por el microprocesador.
- Su estado de ejecución en un momento dado; esto es, los valores de los registros del CPU para dicho programa.
- Su memoria de trabajo, es decir, la memoria que ha reservado y sus contenidos.
- Otra información que permite al sistema operativo su planificación.

Algunas características importantes de los procesos son:

Características	Descripción
el modelo	Se organiza en varios procesos secuenciales (que se ejecutan en multiprogramación)
creación	a) Se inicializa. b) Ejecuta llamadas al sistema. c) El usuario solicita crear proceso. d) Inicio de un trabajo por lotes.
finalización	a) Terminación normal b) Por error (voluntario) c) Error fatal (involuntario) d) Por otro proceso (involuntario)
jerarquía	Se ejecutan según una estructura de prioridades.
estados	a) En ejecución b) Listo (suspendido) c) Bloqueado
implementación	El sistema operativo mantiene un arreglo de estructuras llamada **tabla de procesos**. En esta hay información sobre el estado de procesos, contador de programa, apuntadores de pila, asignación de memoria, etc.

2.2. Comunicación entre procesos

Aunque muchas tareas pueden realizarse en procesos aislados, la gran mayoría requieren la intervención de más de un proceso. Para que dichos procesos cooperantes lleven a buen término una tarea común, es necesario

algún tipo de comunicación entre ellos. Los mecanismos de comunicación entre procesos (I*PC*) habilitan **mecanismos** para que los procesos puedan intercambiar datos y sincronizarse. A la hora de comunicar dos procesos, se consideran dos situaciones diferentes:

- Que los procesos se estén ejecutando en una misma máquina, o bien
- que los procesos se ejecuten en máquinas diferentes.

En el caso de comunicación local entre procesos, aunque existen diferentes mecanismos que se engloban bajo esta denominación común, cada uno de ellos tiene su propósito específico.

Así, las señales son Interrupciones *software* y, como tales, no deberían considerarse parte de la forma habitual de comunicar procesos, y su uso debería restringirse a la comunicación de eventos o situaciones excepcionales. Para dar solución a necesidades de comunicación más genéricas entre procesos, Unix y su adaptación para PC (Linux) utilizan diferentes soluciones que emplean como canal la memoria principal. Una de las primeras formas de este tipo de comunicación son los *pipes* o tuberías. Los *pipes* permiten un mecanismo de comunicación unidireccional sencillo, pero también limitado. Como soluciones más elaboradas a la comunicación entre procesos, se incorporan otros tres tipos de mecanismos: semáforos, memoria compartida y colas de mensajes.

Condiciones de competencia: Las **condiciones de competencia** se dan cuando dos o más procesos intentan acceder a un mismo recurso.

Secciones críticas: Para solucionar las condiciones de competencia se implementó un modelo para prohibir que dos procesos accedan al mismo recurso. El modelo en cuestión se denomina **exclusión mutua**.

Exclusión mutua con espera ocupada: Las soluciones con espera ocupada funcionan de la siguiente manera: cuando un proceso intenta ingresar a su región crítica, verifica si está permitida la entrada. Si no, el proceso se queda esperando hasta obtener el permiso.

Desactivación de interrupciones (activar y desactivar): El método más simple para evitar las condiciones de competencia es hacer que cada proceso desactive todas sus interrupciones antes de entrar a su sección crítica, y las active una vez que salió de la misma. Este modelo, como se puede observar, tiene un gran problema: si se produce una falla mientras que el proceso está en la región crítica, no se puede salir de esta, y el sistema operativo no recuperaría el control.

Semáforos: Un **semáforo** es una estructura diseñada para sincronizar dos o más *threads* o procesos, de modo que su ejecución se realice de forma ordenada y sin conflictos entre ellos.

Se definieron originalmente en 1968 por Dijkstra. Fue presentado como un nuevo tipo de variable.

Es una solución al problema de la exclusión mutua con la introducción del concepto de **semáforo binario**. Esta técnica permite resolver la mayoría de los problemas de sincronización entre procesos, y forma parte del diseño de muchos sistemas operativos y de lenguajes de programación concurrentes.

Un semáforo binario es un indicador (S) de condición que registra si un recurso está disponible o no.

Un semáforo binario solo puede tomar dos valores: 0 y 1. Si para un semáforo binario S = 1, entonces el recurso está disponible y la tarea lo puede utilizar. Si S = 0, el recurso no está disponible, y el proceso debe esperar. Los semáforos se implementan con una cola de tareas o de condición a la cual se añaden los procesos que están en espera del recurso. Solo se permiten tres operaciones sobre un semáforo:

1. Inicializar
2. Espera (*wait*)
3. Señal (*signal*)

Una implementación de semáforos en Java, se muestra en el siguiente código:

```
public class Worker implements Runnable
{
   private Semaphore sem;
   private String name;

  public Worker(Semaphore  sem, String Name)
    {
        this.sem = sem;
        this.name = name;
     }
   public void run()
  {
       while (true) {
           sem.acquire();
           MutualExclusionUtilities.criticalSection(name);
```

```
            sem.release();
            MutualExclusionUtilities.remainderSection(name);
   }
 }
}
```

Implementación AQUIRE()

```
acquire(){
        value--;
           if (value<0) {
                      //agregar este proceso a la lista
                       block;
}
}
```

Implementación de release()

```
release() {
    value++;
  if (value<=0) {
             //remover proceso P de la lista
          wakeup(P);

}
}
```

2.3. Problemas clásicos de comunicación entre procesos

2.3.1. El problema de la cena de los filósofos

Cinco filósofos se sientan alrededor de una mesa, y pasan su vida cenando y pensando. Cada filósofo tiene un plato de fideos y un tenedor a la izquierda de su plato. Pero para comer los fideos son necesarios dos tenedores, y cada filósofo puede tomar el tenedor que esté a su izquierda o derecha, uno por vez (o sea, no puede tomar los dos al mismo tiempo, pero puede tomar uno y después el otro). Si cualquier filósofo coge un tenedor y el otro está ocupado, se quedará esperando, con el tenedor en la mano, hasta que pueda coger el otro tenedor, para luego empezar a comer.

Si dos filósofos adyacentes intentan tomar el mismo tenedor a una vez, se produce una condición de carrera: ambos compiten por tomar el mismo tenedor, y uno de ellos se queda sin comer.

Si todos los filósofos cogen el tenedor que está a su derecha al mismo tiempo, entonces todos se quedarán esperando eternamente, porque alguien debe liberar el tenedor que les falta. Nadie lo hará, porque todos se encuentran en la misma situación (esperando que alguno deje sus tenedores). Entonces, los filósofos se morirán de hambre. Este bloqueo mutuo se denomina *deadlock*.

El problema consiste en encontrar un algoritmo que permita que los filósofos nunca se mueran de hambre.

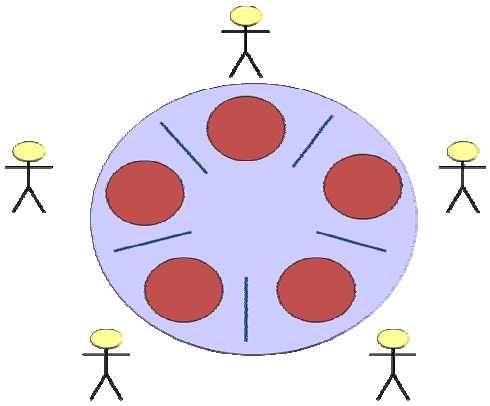


Figura 2: Los cincos filósofos.
Fuente: Gómez Jiménez, Enrique.

El código en Java:

```
import java.util.*;

public class Philosopher implements Runnable {
       private int id;
       private DiningRoom diningRoom;

      public Philosopher(int i, DiningRoom dr) {
              id=i;
              diningRoom=dr;
       }
      public void think() {
                    System.out.println(id+": Pensando!..");
                    try { Thread.sleep(50+(int)(Math.random()*800.0)); }
                    catch (Exception e) { }
          }
       public void run() {
              for (int i=0; i<3; i++) {
                     think();
```

```
                diningRoom.dine(id);
            }
        }

          public static void main(String args[]) {
                DiningRoom dr=new DiningRoom();
                Philosopher p[]={ new Philosopher(0,dr),
                        new Philosopher(1,dr), new Philosopher(2,dr),
                        new Philosopher(3,dr), new Philosopher(4,dr) };
                for (int i=0; i<p.length; i++)
                        new Thread(p[i]).start();
          }
}
```

Implementando la clase DiningRoom:

```
class DiningRoom {

    private final static int THINKING=0;
    private final static int HUNGRY=1;
    private final static int EATING=2;
    private int[] state = {
THINKING,THINKING,THINKING,THINKING,THINKING };

        private void eat(int p) {
                System.out.println(p+": Comiendo!..");
                try { Thread.sleep(50+(int)(Math.random()*800.0)); }
                catch (Exception e) { }
          }
        public void dine(int p) {
              grabForks(p);
              eat(p);
              releaseForks(p);
        }
        private synchronized void grabForks(int p) {
              state[p]=HUNGRY;
                  System.out.println(p+": Esperando! & Hambriento!..");
              test(p);
              while ( state[p] != EATING )
                try { wait(); } catch(Exception e) {}
        }
        private synchronized void releaseForks(int p) {
              state[p]=THINKING;
              test((p+4)%5);
```

```
            test((p+1)%5);
        }
        private void test(int p) {
            if ( state[p] == HUNGRY
                && state[(p+1)%5] != EATING
                && state[(p+4)%5] != EATING
            ) {
                state[p]=EATING;
                notifyAll();
            }
        }
}
```

Creando la clase Philosopher:

```
import Semaphore;

public class Philosopher extends Thread {
   //Compartido por  todos los filósofos
   public final static int N = 5;   // Numero de filósofos


   public final static int THINKING = 0; // Filosofo esta pensando
   public final static int HUNGRY = 1;   // Filosofo esta hambriento
   public final static int EATING = 2;   // Filosofo esta comiendo


  private static int state[] = new int[N]; //Arreglo de estados


 private static Semaphore mutex = new Semaphore(1); //Para exclusion
mutua
                                        //Regiones criticas
 private static Semaphore s[] = new Semaphore[N];
  //Uno por cada filósofo


   // Variable de estancia
   public int myNumber;                 // Cual filosofo soy?
   public int myLeft;                  // A mi izquierda
   public int myRight;                  // A mi derecha


   public Philosopher(int i) {           // Crear un filosofo
```

```
        myNumber = i;
        myLeft = (i+N-1) % N;              // Calculo de la izquierda
        myRight = (i+1) % N;               // Calcula de la derecha
    }


    public void run() {                   // Vamos!
        while(true){
            think();                      // Pensando!
            take_forks();                 // Adquirir dos Tenedores o bloque
            eat();                        // Comer!
            put_forks();        // Ponga los dos tenedores de vuelta sobre la mesa
        }
    }


    public void take_forks(){             // Tome el tenedor
        mutex.down();                     // Entra en la región critica
        state[myNumber] = HUNGRY; // Registra el hecho de que tengo
hambre
        test(myNumber);           // Trate de adquirir dos Tenedores
        mutex.up();             // Entra en región critica
        s[myNumber].down();    // Bloquea si se tienen los dos tenedores
    }

    public void put_forks(){
        mutex.down();                     // Entra en región crítica
        state[myNumber] = THINKING;   // Filosofo terminó de comer
        test(myLeft);       // Ve si el de la izquierda puede comer ahora
        test(myRight);     // Ve si el de la derecha puede comer ahora
        mutex.up();        // Entra en región critica
    }

    public void test(int k){              // Comprueba filosofo K
                                          // desde 0 hasta N-1
        int onLeft = (k+N-1) % N;      // K's Vecino izquierdo
        int onRight = (k+1) % N;       // K's vecino derecho
        if( state[k] == HUNGRY
            && state[onLeft] != EATING
            && state[onRight] != EATING ) {
            // Grab those forks
            state[k] = EATING;
            s[k].up();
        }
```

```
    }

    public void think(){
        System.out.println("Filosofo " + myNumber + " esta pensando!");
        try {
            sleep(1000);
        } catch (InterruptedException ex){ }
    }

    public void eat(){
        System.out.println("Filosofo " + myNumber + " esta comiendo");
        try {
            sleep(5000);
        } catch (InterruptedException ex){ }
    }

    public static void main(String args[]) {

        Philosopher p[] = new Philosopher[N];

        for(int i=0; i<N; i++) {
            // Crea cada filosofo y su semáforo
            p[i] = new Philosopher(i);
            s[i] = new Semaphore(0);

            // Comienza a ejecutar el hilo
            p[i].start();
        }
    }
}
```

2.3.2. El problema de los lectores y escritores

El problema de la cena de los filósofos es útil para modelar procesos que compiten por el acceso exclusivo a un número limitado de recursos, como una unidad de cinta u otro dispositivo de E/S. Otro problema famoso es el de los lectores y escritores (Courtois *et al.*, 1971), que modela el acceso a una base de datos. Supóngase una base de datos, con muchos procesos que compiten por leer y escribir en ella. Se puede permitir que varios procesos lean de la base de datos al mismo tiempo, pero si uno de los procesos está escribiendo (es decir, modificando) la base de datos, ninguno de los demás debería tener acceso a ésta, ni siquiera los lectores. La pregunta es ¿cómo programaría los lectores y escritores?

El código en Java:

La interfaz Bufer.java especifica los métodos llamados por el Productor y el Consumidor:

```
public interface Bufer {
   public void establecer( int valor );  // colocar valor en Bufer
   public int obtener();                 // devolver valor de Bufer
}
```

La clase SalidaRunnable.java actualiza el objeto JTextArea con los resultados:

```
import javax.swing.*;
public class SalidaRunnable implements Runnable {
   private JTextArea areaSalida;
   private String mensajeParaAnexar;
   // inicializar areaSalida y mensaje
   public SalidaRunnable( JTextArea salida, String mensaje )
   {
      areaSalida = salida;
      mensajeParaAnexar = mensaje;
   }
   // método llamado por SwingUtilities.invokeLater para actualizar
areaSalida
   public void run()
   {
      areaSalida.append( mensajeParaAnexar );
   }
} // fin de la clase SalidaRunnable
```

El método run de Consumidor.java controla un subproceso que itera diez veces y lee un valor de ubicacionCompartida cada vez

```
import javax.swing.*;
public class Consumidor extends Thread {
   private Bufer ubicacionCompartida; // referencia al objeto compartido
   private JTextArea areaSalida;
   // constructor
   public Consumidor( Bufer compartido, JTextArea salida )
   {
      super( "Consumidor" );
      ubicacionCompartida = compartido;
      areaSalida = salida;
```

```
  }
  // leer el valor de ubicacionCompartida diez veces y sumar los valores
  public void run()
  {
    int suma = 0;
    for ( int cuenta = 1; cuenta <= 10; cuenta++ ) {
      // estar inactivo de 0 a 3 segundos, leer el valor de Bufer y sumarlo a
suma
      try {
        Thread.sleep( ( int ) ( Math.random() * 3001 ) );
        suma += ubicacionCompartida.obtener();
      }
      // si se interrumpió el subproceso inactivo, imprimir el rastreo de la
pila
      catch ( InterruptedException excepcion ) {
        excepcion.printStackTrace();
      }
    }
    String nombre = getName();
    SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
      "\nTotal que consumió " + nombre + ": " + suma + ".\n" +
      nombre + " terminado.\n ") );
  } // fin del método run
} // fin de la clase Consumidor
```

El método run de Productor.java controla un subproceso que almacena valores de 11 a 20 en ubicacionCompartida:

```
import javax.swing.*;
public class Productor extends Thread {
  private Bufer ubicacionCompartida;
  private JTextArea areaSalida;
  // constructor
  public Productor( Bufer compartido, JTextArea salida )
  {
    super( "Productor" );
    ubicacionCompartida = compartido;
    areaSalida = salida;
  }
  // almacenar valores de 11 a 20 en el búfer de ubicacionCompartida
  public void run()
  {
    for ( int cuenta = 11; cuenta <= 20; cuenta ++ ) {
```

```
        // estar inactivo de 0 a 3 segundos, después colocar valor en Bufer
        try {
          Thread.sleep( ( int ) ( Math.random() * 3000 ) );
          ubicacionCompartida.establecer( cuenta );
        }
        // si se interrumpió el subproceso inactivo, imprimir el rastreo de la
pila
        catch ( InterruptedException excepcion ) {
          excepcion.printStackTrace();
        }
      }
      String nombre = getName();
      SwingUtilities.invokeLater( new SalidaRunnable( areaSalida, "\n" +
        nombre + " terminó de producir.\n" + nombre + " terminado.\n" ) );
    } // fin del método run
} // fin de la clase Productor
```

BuferCircular.java sincroniza el acceso a un arreglo de búferes compartidos

```
import javax.swing.*;
public class BuferCircular implements Bufer {
  // cada elemento del arreglo es un búfer
  private int buferes[] = { -1, -1, -1 };
  // cuentaBuferesOcupados mantiene la cuenta de buferes ocupados
  private int cuentaBuferesOcupados = 0;
  // variables que mantienen las ubicaciones de lectura y escritura en el
búfer
  private int ubicacionLectura = 0, ubicacionEscritura = 0;
  // referencia al componente de la GUI que muestra la salida
  private JTextArea areaSalida;
  // constructor
  public BuferCircular( JTextArea salida )
  {
    areaSalida = salida;
  }
  // colocar valor en búfer
  public synchronized void establecer( int valor )
  {
    // obtener nombre del subproceso que llamó a este método, para
mostrarlo en pantalla
    String nombre = Thread.currentThread().getName();
    // mientras no haya ubicaciones vacías, colocar subproceso en estado de
espera
```

```
      while ( cuentaBuferesOcupados == buferes.length ) {
        // mostrar información de subproceso y de búfer, después esperar
        try {
          SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
            "\nTodos los buferes llenos. " + nombre + " espera." ) );
          wait();
        }
        // si se interrumpió el proceso en espera, imprimir el rastreo de la pila
        catch ( InterruptedException excepcion )
        {
          excepcion.printStackTrace();
        }
      } // fin de  instrucción while
      // colocar valor en ubicacionEscritura de buferes
      buferes[ ubicacionEscritura ] = valor;
      // actualizar componente de la GUI de Swing con el valor producido
      SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
        "\n" + nombre + " escribe " + buferes[ ubicacionEscritura ] + " ") );
      // acaba de producir un valor, por lo que se incrementa el número de
buferes ocupados
      ++cuentaBuferesOcupados;
      // actualizar ubicacionEscritura para la siguiente operación de escritura
      ubicacionEscritura = ( ubicacionEscritura + 1 ) % buferes.length;
      // mostrar contenido de buferes compartidos
      SwingUtilities.invokeLater( new SalidaRunnable(
        areaSalida, crearSalidaEstado() ) );
      notify(); // regresar el subproceso en espera (si hay uno) al estado listo
    } // fin del método establecer
    // devolver valor de búfer
    public synchronized int obtener()
    {
      // obtener nombre del subproceso que llamó a este método, para
mostrarlo en pantalla
      String nombre = Thread.currentThread().getName();
      // mientras no haya datos que leer, colocar el subproceso en estado de
espera
      while ( cuentaBuferesOcupados == 0 ) {
        // mostrar información de subproceso y de búfer, después esperar
        try {
          SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
            "\nTodos los buferes vacíos. " + nombre + " espera.") );
          wait();
        }
```

```
        // si se interrumpió el subproceso en espera, imprimir el rastreo de la
pila
        catch ( InterruptedException excepcion ) {
          excepcion.printStackTrace();
        }
      } // fin de instrucción while
      // obtener valor en ubicacionLectura actual
      int valorLectura = buferes[ ubicacionLectura ];
      // actualizar componente de la GUI de Swing con el valor consumido
      SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
        "\n" + nombre + " lee " + valorLectura + " ") );
      // acaba de consumir un valor, por lo que se decrementa el número de
buferes ocupados
      --cuentaBuferesOcupados;
      // actualizar ubicacionLectura para la siguiente operación de lectura
      ubicacionLectura = ( ubicacionLectura + 1 ) % buferes.length;
      // mostrar contenido de buferes compartidos
      SwingUtilities.invokeLater( new SalidaRunnable(
        areaSalida, crearSalidaEstado() ) );
      notify(); // regresar el subproceso en espera (si hay uno) al estado listo
      return valorLectura;
    } // fin del método obtener
    // crear salida de estado

    public String crearSalidaEstado()
    {
      // primera línea de información de estado
      String salida =
        "(buferes ocupados: " + cuentaBuferesOcupados + ")\nbuferes: ";
      for ( int i = 0; i < buferes.length; i++ )
        salida += " " + buferes[ i ] + "  ";
      // segunda línea de información de estado
      salida += "\n         ";
      for ( int i = 0; i < buferes.length; i++ )
        salida += "---- ";
      // tercera línea de información de estado
      salida += "\n         ";
      // anexar indicadores de ubicacionLectura (L) y ubicacionEscritura (E)
      // debajo de las ubicaciones de búfer apropiadas
      for ( int i = 0; i < buferes.length; i++ )
        if ( i == ubicacionEscritura &amp;amp;&amp;amp; ubicacionEscritura ==
ubicacionLectura )
          salida += " EL  ";
        else if ( i == ubicacionEscritura )
```

```
            salida += " E   ";
         else if ( i == ubicacionLectura )
            salida += "  L  ";
         else
            salida += "     ";
      salida += "\n";
      return salida;
   } // fin del método crearSalidaEstado
} // fin de la clase BuferCircular
```

PruebaBuferCircular.java muestra a dos subprocesos manipulando un búfer circular:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
// establecer los subprocesos productor y consumidor e iniciarlos
public class PruebaBuferCircular extends JFrame {
   JTextArea areaSalida;
   // configurar la GUI
   public PruebaBuferCircular()
   {
      super( "Demostración de sincronización de subprocesos" );
      areaSalida = new JTextArea( 20,30 );
      areaSalida.setFont( new Font( "Monospaced", Font.PLAIN, 12 ) );
      getContentPane().add( new JScrollPane( areaSalida ) );
      setSize( 350, 500 );
      setVisible( true );
      // crear objeto compartido utilizado por los subprocesos; utilizamos una
referencia
      // BuferCircular en vez de una referencia Bufer, para poder invocar al
      // método crearSalidaEstado de BuferCircular
      BuferCircular ubicacionCompartida = new BuferCircular( areaSalida );
      // mostrar el estado inicial de los búferes en BuferCircular
      SwingUtilities.invokeLater( new SalidaRunnable( areaSalida,
         ubicacionCompartida.crearSalidaEstado() ) );
      // establecer subprocesos
      Productor productor = new Productor( ubicacionCompartida, areaSalida
);
      Consumidor consumidor = new Consumidor( ubicacionCompartida,
areaSalida );
      productor.start();  // iniciar subproceso productor
      consumidor.start();  // iniciar subproceso consumidor
   } // fin del constructor
```

```
  public static void main ( String args[] )
  {
    PruebaBuferCircular aplicacion = new PruebaBuferCircular();
    aplicacion.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
  }
} // fin de la clase PruebaBuferCircular
```

2.3.3. El problema del barbero dormilón

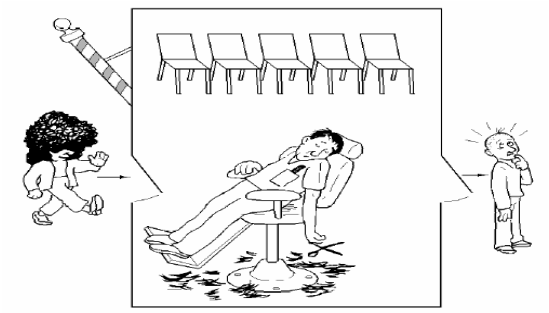


Figura 3: El barbero dormilón.
Fuente: Tanenbaum (2003: 130).

Otro de los problemas clásicos de este paradigma ocurre en una peluquería. Esta tiene un barbero, una silla de peluquero y **n** sillas para que se sienten los clientes en espera. Si no hay clientes presentes, el barbero se sienta y se duerme. Cuando llega un cliente, este debe despertar al barbero dormilón. Si llegan más clientes mientras el barbero corta el cabello de un cliente, ellos se sientan (si hay sillas desocupadas) o, en todo caso, salen de la peluquería. El problema consiste en programar al barbero y los clientes sin entrar *en condiciones de competencia.* Una solución sería: cuando el barbero abre su negocio por la mañana, ejecuta el procedimiento *barber*, lo que establece un bloqueo en el semáforo *customers,* hasta que alguien llega; después se va a dormir. Cuando llega el primer cliente el ejecuta *customer*. Si otro cliente llega poco tiempo después, el segundo no podrá hacer nada. Luego verifica entonces si el número de clientes que esperan es menor que el número de sillas; si esto no ocurre, sale sin su corte de pelo. Si existe una silla disponible, el cliente incrementa una variable contadora. Luego realiza un up en el semáforo *customer,* con lo que despierta al barbero. En este momento, tanto el cliente como el barbero están despiertos; luego, cuando le toca su turno al cliente, le cortan el pelo.

A continuación se presenta una implementación de este problema en Java:

```
        // Suposiciones:
       // - El barbero corta el pelo fuera del objeto protegido
       //   Si lo cortara dentro, sería menos realista la simulación
       //   Del tiempo en que se tarda en hacer esta operación. Si
       //   No se hace este retardo, es decir si el tiempo de corte
       //   De pelo fuera prácticamente 0, no habría casi nunca
       //   Procesos esperando.
       // - Se simula la silla del barbero y las sillas de la sala
       //   De espera.

public class barberia {
       private int nSillasEspera;
       private int nSillasEsperaOcupadas = 0;
       private boolean sillaBarberoOcupada = false;
       private boolean finCorte = false;
       private boolean barberoDormido = false;

       //JAVA: sólo puede haber N_Sillas_Espera_max hebras
       //esperando dentro del monitor a que le toque.

         public barberia(int nSillasEspera)         {
              this.nSillasEspera = nSillasEspera;   }

         public synchronized boolean entrar(int clienteId)

            throws InterruptedException {

          if (nSillasEsperaOcupadas == nSillasEspera) {
                // Si no hay sillas libres, me voy sin cortar el pelo
              System.out.println("---- El cliente " + clienteId + " se va sin
cortarse el pelo");
                return false; }
          else {
                //Me quedo esperando si la silla del barbero está ocupada
                nSillasEsperaOcupadas ++;
                System.out.println ("---- El cliente " + clienteId + " se sienta en
la silla de espera");
                while (sillaBarberoOcupada) {wait();}

                //Desocupo la silla de espera
                nSillasEsperaOcupadas --;
```

```
            //Me siento en la silla del barbero
            sillaBarberoOcupada = true;
            finCorte = false;

            //Si el barbero está dormido lo despierto
            if (barberoDormido) {
                  System.out.println
                      ("---- El cliente " + clienteId + " despierta al barbero");
                  notifyAll();}

            //Espero a que me corte el pelo
            System.out.println("---- El cliente " + clienteId + " en la silla de
barbero");
            while (!finCorte) {wait();}
            sillaBarberoOcupada = false;
             //Que pase el siguiente
            notifyAll();
            System.out.println("---- El cliente " + clienteId + " se va con el
pelo cortado");
            return true;

        }
        }

         public synchronized void esperarCliente()
           throws InterruptedException {
            //El barbero espera a que llegue un cliente
            //Se supone que le corta el pelo fuera del monitor
            barberoDormido = true;
            while(!sillaBarberoOcupada) {
              System.out.println("++++ Barbero esperando cliente");
                  wait();
                  }
            barberoDormido = false;
            System.out.println("++++ Barbero cortando el pelo");
         }

         public synchronized void acabarCorte(){
            finCorte = true;
             System.out.println("++++ Barbero termina de cortar el pelo");
            notifyAll();
         }
}
```

2.4. Calendarización

El planificador es un componente funcional muy importante de los sistemas operativos multitarea y multiproceso. Es esencial en los sistemas operativos de tiempo real. Su función consiste en repartir el tiempo disponible de un microprocesador entre todos los procesos que están disponibles para su ejecución. A continuación, se presenta un código Java para la ejecución de procesos según prioridad.

```
import java.awt.*;
import java.applet.Applet;

// En este applet se crean dos threads que incrementan un contador, se
// Proporcionan distintas prioridades a cada uno y se para cuando los dos
coinciden

public class SchThread extends Applet {
   Contar alto,bajo;

   public void init() {
      // Creamos un thread en 200, ya adelantado
      bajo = new Contar( 200 );
      // El otro comienza desde cero
      alto = new Contar( 0 );
      // Al que comienza en 200 le asignamos prioridad mínima
      bajo.setPriority( Thread.MIN_PRIORITY );
      // Y al otro máxima
      alto.setPriority( Thread.MAX_PRIORITY );
      System.out.println( "Prioridad alta es "+alto.getPriority() );
      System.out.println( "Prioridad baja es "+bajo.getPriority() );
      }

   // Arrancamos los dos threads, y vamos repintando hasta que el thread
   // que tiene prioridad más alta alcanza o supera al que tiene prioridad
   // más baja, pero empezó a contar más alto
   public void start() {
      bajo.start();
      alto.start();
      while( alto.getContar() < bajo.getContar() )
         repaint();
      repaint();
      bajo.stop();
      alto.stop();
      }
```

```
    // Vamos pintando los incrementos que realizan ambos threads
    public void paint( Graphics g ) {
        g.drawString( "bajo = "+bajo.getContar()+
            " alto = "+alto.getContar(),10,10 );
        System.out.println( "bajo = "+bajo.getContar()+
            " alto = "+alto.getContar() );
        }

    // Para parar la ejecución de los threads
    public void stop() {
        bajo.stop();
        alto.stop();
        }
    }

//---------------------------------------- Final del fichero SchThread.java
```

Capítulo 3: Recursos

3.1. Recursos

Los **recursos en sistemas operativos** pueden ser:

- *hardware*: un dispositivo de E/S
- *software*: información (BD, variables, etc.)

Estos recursos se agrupan en:

3.1.1. Recursos expropiables y no expropiables

Recurso expropiable

Es un recurso al que se le puede quitar a quien lo tiene sin causarle daño.
Ejemplo: La memoria se puede expropiar, respaldándosela en disco a quien la tenía.

Recurso no expropiable

No se puede quitar a su dueño sin hacer que su cómputo falle.
Ejemplo: Impresora, quemadora de CDs, etc.

3.2. Introducción a los bloqueos irreversibles

Bloqueo irreversible (BI)

- Un conjunto de procesos cae en un BI si cada proceso del conjunto está esperando un suceso que solo otro proceso del conjunto puede causar.
- Hay BI en el nivel proceso o en el nivel computadora en una red

3.3. Algoritmo del avestruz

Una alternativa para enfrentar el problema del interbloqueo de procesos es a través del algoritmo del avestruz. A esta estrategia se la denomina

algoritmo del avestruz (*ostrich*): esconder la cabeza en la tierra y pretender que no existe problema alguno. La justificación de este método es que, si el interbloqueo se presenta con una frecuencia baja en comparación con los fallos del sistema por otras razones (errores en el sistema operativo, fallos en el *hardware*, etc.), no tiene sentido tomar medidas para evitar el problema a costa de reducir el rendimiento del sistema. Un ejemplo es el sistema operativo Unix.

3.4. Detección de bloques irreversibles y recuperación posterior

Un aspecto importante es la frecuencia de detección de interbloqueos, que suele ser un parámetro del sistema. Una posibilidad extrema es comprobar el estado cada vez que se solicita un recurso y este no puede ser asignado. El inconveniente es el tiempo de CPU que se utiliza. Otra alternativa es activar el algoritmo ocasionalmente, a intervalos regulares, o cuando uno o más procesos queden bloqueados durante un tiempo sospechosamente largo.

Una frecuencia alta de comprobación tiene la ventaja de descubrir con prontitud los interbloqueos, capacitando al sistema a actuar rápidamente. Con una frecuencia baja se reduce el gasto de tiempo de ejecución para la detección, a expensas de dejar interbloqueos sin detectar durante más largos períodos. Esta estrategia puede redundar en un empobrecimiento en la utilización de recursos, puesto que los procesos interbloqueados continúan reteniendo los recursos ya concedidos sin realizar trabajo alguno con ellos.

Para romper el bloqueo de un sistema, hay que anular una o más de las condiciones necesarias para el bloqueo. Normalmente, varios procesos **perderán** algo o todo lo realizado hasta el momento.

Los principales factores que dificultan la recuperación del bloqueo son los siguientes:

- Puede no estar claro si el sistema se ha bloqueado o no.
- Muchos sistemas tienen limitaciones para suspender un proceso por tiempo indefinido y reanudarlo más tarde:
  - Ejemplo: Los procesos de tiempo real, que deben funcionar continuamente, no son fáciles de suspender y reanudar.
- Los procedimientos de suspensión / reanudación implican una sobrecarga considerable.
- La **sobrecarga de recuperación está en función de la magnitud del bloqueo** (algunos, decenas o centenas de procesos involucrados).

Generalmente, la **recuperación** suele realizarse:

- Retirando forzosamente (cancelando) un proceso.
- Reclamando sus recursos.
- Permitiendo que los procesos restantes puedan finalizar.

Los procesos pueden ser retirados (cancelados) de acuerdo con un **orden de prioridades**, existiendo las siguientes dificultades:

- Pueden no existir las prioridades de los procesos bloqueados.
- Las prioridades instantáneas (en un momento dado) pueden ser incorrectas o confusas debido a consideraciones especiales; por ejemplo, procesos de baja prioridad que tienen prioridad alta momentáneamente debido a un tiempo tope inminente.
- La decisión óptima puede requerir un gran esfuerzo.

Algunas **formas de recuperación** ante bloqueos son:

- **Recuperación mediante la apropiación**
- **Recuperación mediante *rollback***
- **Recuperación mediante la eliminación de procesos**

3.5. Cómo evitar los bloqueos irreversibles

La idea básica de la evitación de interbloqueos es conceder únicamente las peticiones de recursos disponibles que no conduzcan a estados propensos al interbloqueo. En general, existe un "asignador" de recursos que, ante una petición determinada, examina el efecto de conceder dicha petición. Si la concesión puede conducir a un estado potencialmente peligroso, el proceso solicitante queda suspendido hasta el momento en que su petición pueda ser concedida sin problemas, lo cual suele suceder después de que uno o más de los recursos retenidos por otros procesos hayan sido liberados.

Un algoritmo de evitación examina dinámicamente el estado de asignación de recursos, para asegurar que no pueda presentarse la condición de espera circular. El estado de asignación de recursos viene definido por el número de recursos disponibles y asignados, y por la demanda máxima de los procesos. Los principales algoritmos para evitar interbloqueos se basan en el concepto de **estado seguro**.

En la siguiente figura se muestra un modelo para dos procesos (*P1* y *P2*) y dos recursos (una impresora y un *plotter*). El eje horizontal representa el número de instrucciones ejecutadas por el proceso *P1* y el eje vertical el

número de instrucciones ejecutadas por el proceso *P2*. En el instante *I1*, *P1* solicita la impresora, en *I2* necesita el *plotter*. La impresora y el *plotter* se liberan en los instantes *I3* e *I4*, respectivamente. El proceso *P2* necesita el *plotter* en el intervalo de *I5* hasta *I7,* y la impresora desde *I6* hasta *I8*.

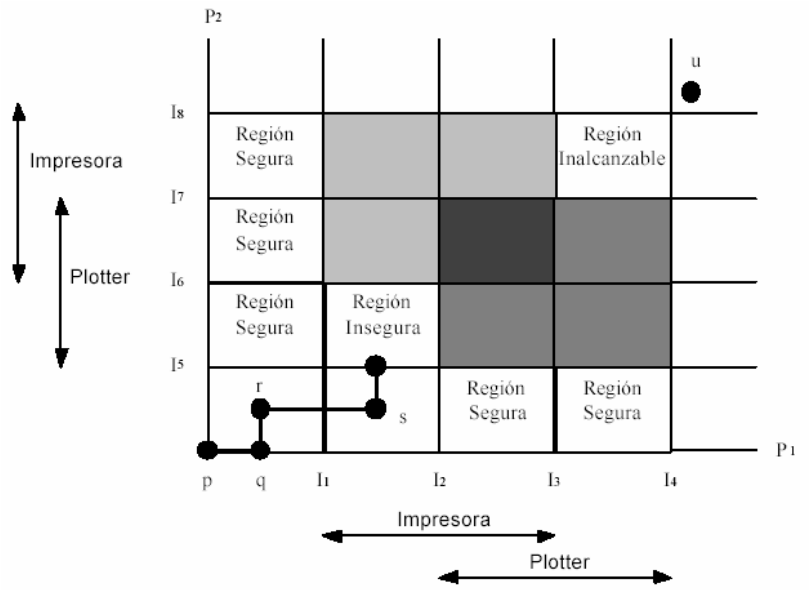


Figura 4: Gráfico de asignación de recursos con un interbloqueo.
Fuente: Tanenbaum (2003: 176).

Una línea gruesa representa una posible evolución (trayectoria). Cada punto representa un estado conjunto de los dos procesos. Los tramos horizontales indican ejecución de *P1*, los verticales, ejecución de *P2*. Con un único procesador, todas las trayectorias serán horizontales o verticales. Además, el movimiento siempre será hacia arriba o hacia la derecha (los procesos no pueden retroceder).

Estados seguros e inseguros

Un estado es seguro si el sistema puede asignar recursos a cada proceso hasta alcanzar el máximo de sus necesidades siguiendo algún orden arbitrario, y aún así evitar el interbloqueo. Más normalmente, un sistema se encuentra en estado seguro solo si existe una secuencia segura. *<P1, P2, ..., Pn>* es segura para el estado actual de asignación si, para cada proceso *Pi*, los recursos que aún puede solicitar *Pi* pueden satisfacerse con los recursos actualmente disponibles más los retenidos por todos los *Pj*, tales que *j* < *i*. En esta situación, si los recursos que necesita *Pi* no están inmediatamente disponibles, entonces *Pi* puede esperar a que terminen todos los procesos que le preceden en la secuencia. Una vez terminados éstos, *Pi* puede obtener todos los recursos necesarios, completar su tarea, devolver los recursos que se le han asignado y terminar.

Cuando *Pi* termina, *Pi+1* puede obtener los recursos que necesita, y así sucesivamente. Si no existe esta secuencia, se dice que **el estado es inseguro**. Un estado inseguro no es un estado de interbloqueo, pero un estado de interbloqueo sí es un estado inseguro. Es decir, no todos los estados inseguros lo son de interbloqueo. En la siguiente figura se puede observar que el conjunto de estados de interbloqueo es un subconjunto del de estados inseguros. Un estado inseguro puede conducir a un interbloqueo. Mientras el estado del sistema sea seguro, el sistema operativo puede evitar estados inseguros y, por tanto, de interbloqueo.

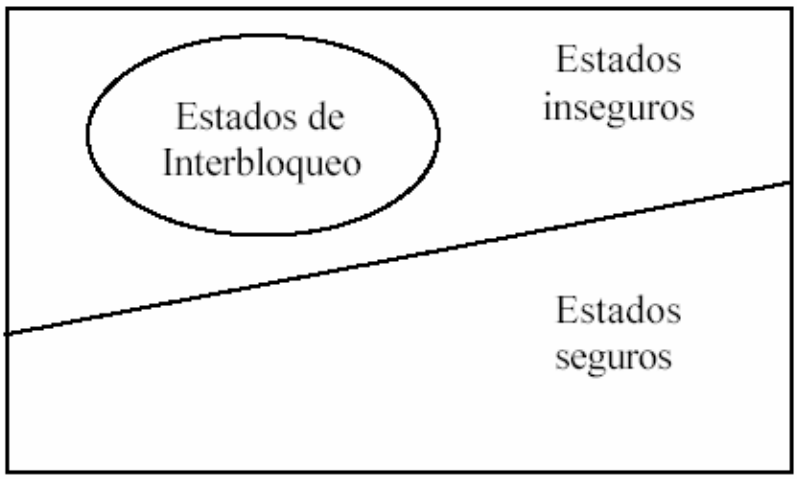


Figura 5: Tipos de estados.
Fuente: Gómez Jiménez, Enrique.

3.5.3. El algoritmo del banquero de un solo recurso

El algoritmo del banquero en sistemas operativos es una forma de evitar el interbloqueo, propuesta por primera vez por Edsger Dijkstra. Es un acercamiento teórico para evitar los interbloqueos en la planificación de recursos. Requiere conocer con anticipación los recursos que serán utilizados por todos los procesos. Esto último generalmente no puede ser satisfecho en la práctica.

Es un **algoritmo de planificación** que puede **evitar los bloqueos**:

En la analogía:

- Los clientes son los **procesos**; las unidades de crédito son los **recursos** del sistema, y el banquero es el *SO*.

- El banquero sabe que no todos los clientes necesitarán su crédito máximo otorgado en forma inmediata. Por ello, reserva menos unidades (recursos) de las totales necesarias para dar servicio a los clientes.

Un estado inseguro no tiene que llevar a un bloqueo*.*

El algoritmo del banquero consiste en:

- Estudiar cada solicitud al ocurrir esta.

- Ver si su otorgamiento conduce a un **estado seguro***:*
    - En caso positivo, se otorga la solicitud.
    - En caso negativo, se la pospone.

- Para ver si un estado es seguro:
    - Verifica si tiene los recursos suficientes para satisfacer a otro cliente:
        - En caso afirmativo, se supone que los préstamos se pagarán.
        - Se verifica al siguiente cliente cercano al límite, y así sucesivamente.
    - Si en cierto momento se vuelven a pagar todos los créditos, el estado es seguro, y la solicitud original debe ser aprobada.

Este algoritmo usualmente es explicado usando la analogía con el funcionamiento de un banco. Los clientes representan a los procesos, que tienen un crédito límite, y el dinero representa a los recursos. El banquero es el sistema operativo.

El banco confía en que no tendrá que permitir a todos sus clientes la utilización de todo su crédito a la vez. El banco también asume que si un cliente maximiza su crédito, será capaz de terminar sus negocios y retornar el dinero de vuelta a la entidad, permitiendo servir a otros clientes.

3.5.3. El algoritmo del banquero para múltiples recursos

Acá también los procesos deben establecer sus **necesidades totales de recursos antes de su ejecución**; y, dada una matriz de **recursos asignados**, el SO debe poder calcular en cualquier momento la matriz de **recursos necesarios** (ver la tabla siguiente).

Recursos Asignados					Recursos Necesarios					
A	3	0	1	1	A	1	1	0	0	
B	0	1	0	0	B	0	1	1	2	E=(6 3 4 2)
C	1	1	1	0	C	3	1	0	0	P=(5 3 2 2)
D	1	1	0	1	D	0	0	1	0	A=(1 0 2 0)
E	0	0	0	0	E	2	1	1	0	
↑										
Procesos										

Figura 6: Ejemplo del algoritmo del banquero.
Fuente: Gómez Jiménez, Enrique.

Se dispone de:

- *"E": vector de recursos existentes*
- *"P": vector de recursos poseídos*
- *"A": vector de recursos disponibles*

El algoritmo para determinar si un *estado es seguro* es el siguiente:

1. Se busca un renglón *"R"* cuyas necesidades de recursos no satisfechas sean menores o iguales que *"A":*
    - Si no existe tal renglón, el sistema se bloqueará en algún momento, y ningún proceso podrá concluirse.
2. Supongamos que el proceso del renglón elegido solicita todos los recursos que necesita y concluye:
    - Se señala el proceso como concluido y se añaden sus recursos al vector *"A".*
3. Se repiten los pasos 1 y 2:
    - Hasta que todos los procesos queden señalados como concluidos, en cuyo caso el estado inicial era seguro, o
    - Hasta que ocurra un bloqueo, en cuyo caso no lo era.

3.6. Prevención de bloqueos irreversibles

Si se puede garantizar que al menos una de las cuatro condiciones de Coffman para el bloqueo nunca se satisface, entonces los bloqueos serán imposibles por razones estructurales (enunciado de Havender).

Havender sugirió las siguientes **estrategias para evitar varias de las condiciones de bloqueo**:

- Cada proceso:
  - Deberá pedir todos sus recursos requeridos de una sola vez.
  - No podrá proceder hasta que le hayan sido asignados.

- Si a un proceso que mantiene ciertos recursos se le niega una nueva petición, este proceso deberá:
  - Liberar sus recursos originales.
  - En caso necesario, pedirlos de nuevo junto con los recursos adicionales.

- Se impondrá la ordenación lineal de los tipos de recursos en todos los procesos:
  - Si a un proceso le han sido asignados recursos de un tipo dado, en lo sucesivo solo podrá pedir aquellos recursos de los tipos que siguen en el ordenamiento.

Havender no presenta una estrategia contra el uso exclusivo de recursos por parte de los procesos, pues se desea permitir el uso de recursos dedicados.

3.6.1. Cómo atacar la condición de exclusión mutua

Si ningún recurso se asignara de manera exclusiva a un solo proceso, nunca tendríamos bloqueos, pero esto es imposible de aplicar, en especial en relación con ciertos tipos de recursos, que en un momento dado no pueden ser compartidos (por ejemplo, impresoras).

Se debe:

- Evitar la asignación de un recurso cuando no sea absolutamente necesario.
- Intentar asegurarse de que los menos procesos posibles puedan pedir el recurso.

3.6.2. Cómo atacar la condición de retener y esperar

Si se puede **evitar que los procesos que conservan recursos esperen más recursos**, se pueden eliminar los bloqueos.

Una forma es exigir a *todos* **los procesos que soliciten todos los recursos** antes de iniciar su ejecución. Si un proceso no puede disponer de todos los recursos, deberá esperar, pero sin retener recursos afectados.

Un problema es que muchos procesos no **saben** el número de recursos necesarios hasta iniciar su ejecución.

Otro problema es que puede significar desperdicio de recursos, dado que todos los recursos necesarios para un proceso están afectando a este desde su inicio hasta su finalización.

Otro criterio aplicable consiste en:

- Exigir a un proceso que solicita un recurso que libere en forma temporal los demás recursos que mantiene en ese momento.
- Hacer que el proceso intente luego recuperar todo al mismo tiempo.

3.6.3. Cómo atacar la condición de no expropiación

Una de las estrategias de Havender requiere que, cuando a un proceso que mantiene recursos le es negada una petición de recursos adicionales, deberá liberar sus recursos y, si es necesario, pedirlos de nuevo junto con los recursos adicionales.

La implementación de esta estrategia **niega la condición de "no apropiación" y los recursos pueden ser retirados** de los procesos que los retienen **antes de la terminación de los procesos***.*

El problema consiste en que el retiro de ciertos recursos de un proceso **puede significar***:*

- La pérdida del trabajo efectuado hasta ese punto.
- La necesidad de repetirlo luego.

Una consecuencia es la **posible postergación indefinida** de un proceso.

3.6.4. Cómo atacar la condición de espera circular

Una forma de atacarla es que un proceso solo está autorizado a **utilizar un recurso en cada momento**:

- Si necesita otros recursos, debe liberar el primero.
- Esto resulta inaceptable para muchos procesos.

Otra forma es la siguiente:

- Todos los recursos se numeran globalmente.
- Los procesos pueden solicitar los recursos en cualquier momento:
    - Las solicitudes se deben hacer según un cierto orden numérico (creciente) de recurso. Debido a esto, la gráfica de asignación de recursos no tendrá ciclos.
- En cada instante uno de los recursos asignados tendrá el número más grande:
    - El proceso que lo posea no pedirá un recurso ya asignado.
    - El proceso terminará o solicitará recursos con números mayores, que estarán disponibles:
        - Al concluir liberará sus recursos.
        - Otro proceso tendrá el recurso con el número mayor y también podrá terminar.
        - Todos los procesos podrán terminar y no habrá bloqueo.

Una **variante** consiste en **eliminar el requisito de adquisición de recursos en orden creciente***:*

- Ningún proceso debe solicitar un recurso con número menor al que posee en el momento.

El problema es que en casos reales podría resultar imposible encontrar un orden que satisfaga a todos los procesos.

3.7.1. Bloqueo de dos fases

Una **operación frecuente en sistemas de bases de datos** consiste en:

- Solicitar el cierre de varios registros.
- Actualizar todos los registros cerrados.
- Ante la ejecución de varios procesos al mismo tiempo, existe un grave riesgo de bloqueo.

El método de la **cerradura de dos fases** consiste en:

- **Primera fase:** El proceso intenta cerrar todos los registros necesarios, uno a la vez.
- **Segunda fase:** Se actualiza y se liberan las cerraduras.
- Si durante la primer fase se necesita algún registro ya cerrado:
    - El proceso libera todas las cerraduras y comienza en la primera fase nuevamente.

- Generalmente esto no resulta aplicable en la realidad:
  - No resulta aceptable dejar un proceso a la mitad y volver a comenzar.
  - El proceso podría haber actualizado archivos, enviado mensajes en la red, etc.

3.7.2. Bloqueos irreversibles que no son por recursos

Los bloqueos también pueden aparecer en **situaciones que no están relacionadas con los recursos**.

Puede ocurrir que dos procesos se bloqueen en espera de que el otro realice cierta acción, por ejemplo, operaciones efectuadas sobre semáforos (indicadores o variables de control) en orden incorrecto.

3.7.3. Inanición

En un sistema dinámico permanentemente hay solicitudes de recursos.

Se necesita un criterio (política) para decidir:

- **Quién** obtiene cuál recurso.
- En **qué** momento.

Podría suceder que ciertos procesos ***nunca lograran el servicio***, aún sin estar bloqueados, porque se privilegia en el uso del recurso a otros procesos.

La **inanición se puede evitar mediante el criterio de asignación de recursos *FIFO*** (el primero en llegar es el primero en despachar (ser atendido).

El proceso que ha esperado el máximo tiempo se despachará a continuación:

- En el transcurso del tiempo, cualquiera de los procesos dados:
  - Será el más antiguo.
  - Obtendrá el recurso necesario.

Capítulo 4: Administración de la memoria

4.1. Administración de memoria básica

Los sistemas operativos administran la memoria trayendo y llevando procesos entre esta y el disco mediante paginación. Pero hay otros que no realizan esta función. Si la memoria principal es insuficiente, entonces se produce la paginación para garantizar suficiente espacio de almacenamiento para la ejecución de procesos.

4.1.1. Monoprogramación sin intercambio, ni paginación

Esta organización facilita la programación de una aplicación al dividirla en dos o más procesos. Además, ofrece la capacidad de tener más de un proceso a la vez en memoria; así puede ofrecer servicios a varios usuarios a la vez.

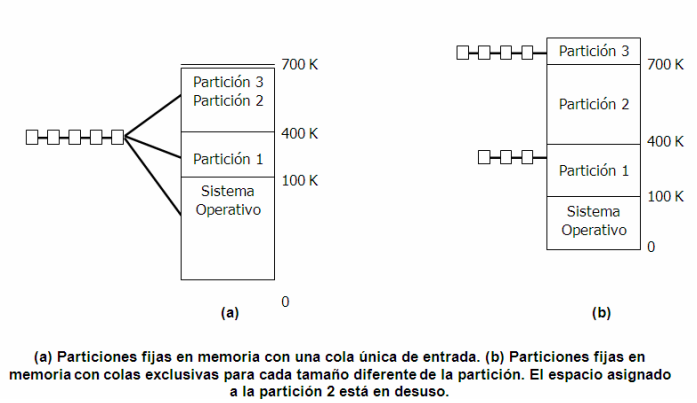
El esquema de **multiprogramación** incrementa el aprovechamiento del CPU, a diferencia de la **monoprogramación**. En esta, solo un proceso reside en memoria a la vez, limitando el uso del procesador a las llamadas que requiera dicho proceso, desperdiciando un promedio del 80% del tiempo del procesador. En cambio, la multiprogramación, al tener varios procesos en la memoria principal y dividiéndose el tiempo de uso del procesador, logra reducir drásticamente el desperdicio del procesador.

4.1.2. Multiprogramación con particiones fijas

Para poder implementar la multiprogramación, se puede hacer uso de particiones fijas o variables en la memoria. **En el caso de las particiones fijas**, la memoria se puede organizar dividiéndose en diversas partes, las cuales pueden variar en tamaño. Esta partición la puede hacer el usuario en forma manual, al iniciar una sesión con la máquina.

Una vez implementada la partición, hay dos maneras de asignar los procesos a ella. La primera es mediante el uso de una cola única (figura **a**) que asigna los procesos a los espacios disponibles de la memoria en la medida en que se vayan desocupando. El tamaño del hueco de memoria disponible es usado para localizar en la cola el primer proceso que quepa en él. Otra forma de asignación es buscar en la cola el proceso de tamaño mayor que se ajuste al hueco. Sin embargo, hay que tomar en cuenta que tal método discrimina a

los procesos más pequeños. Este problema podría tener solución si se asigna una partición pequeña en la memoria en el momento de hacer la partición inicial, el cual sería exclusivo para procesos pequeños.



(a) Particiones fijas en memoria con una cola única de entrada. (b) Particiones fijas en memoria con colas exclusivas para cada tamaño diferente de la partición. El espacio asignado a la partición 2 está en desuso.

Figura 7: Particiones fijas para memoria.
Fuente: Trejo Ramírez (2002: 3).

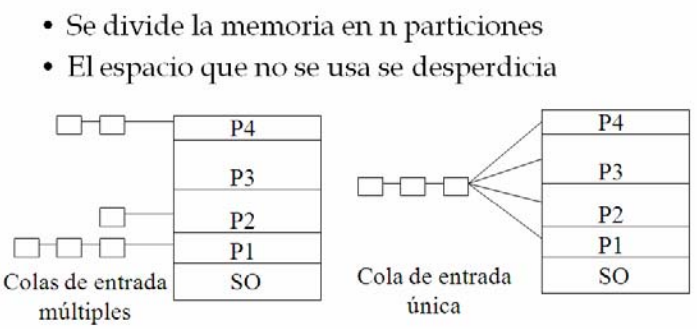


Figura 8: Particiones fijas para colas.
Fuente: Trejo Ramírez (2002: 3).

Esta idea nos lleva a la implementación de otro método para particiones fijas, que es el uso de diferentes colas independientes (figura **b**) exclusivas para cierto rango en el tamaño de los procesos. De esta manera, al llegar un proceso, este sería asignado a la cola de tamaño más pequeño que la pueda aceptar. La desventaja en esta organización es que, si una de las colas tiene una larga lista de procesos en espera, mientras otra cola está vacía, el sector de memoria asignado para ese tamaño de procesos estaría desperdiciándose.

4.2. Intercambio

Este esquema fue originalmente usado por el sistema operativo IBM OS/360 (llamado MFT), el cual ya no está en uso.

El sistema operativo lleva una tabla indicando cuáles partes de la memoria están disponibles y cuáles están ocupadas. Inicialmente, toda la memoria está disponible para los procesos de usuario, y es considerado como un gran bloque o hueco único de memoria. Cuando llega un proceso que necesita memoria, buscamos un hueco lo suficientemente grande para el proceso. Si encontramos uno, se asigna únicamente el espacio requerido, manteniendo el resto disponible para futuros procesos que requieran de espacio.

4.2.1. Administración de memoria con mapas de bits

Este tipo de administración divide la memoria en unidades de asignación, las cuales pueden ser tan pequeñas como unas cuantas palabras o tan grandes como varios kilobytes. A cada unidad de asignación le corresponde un bit en el mapa de bits, el cual toma el valor de 0 si la unidad está libre y 1 si está ocupada (o viceversa). La siguiente figura muestra una parte de la memoria y su correspondiente mapa de bits.

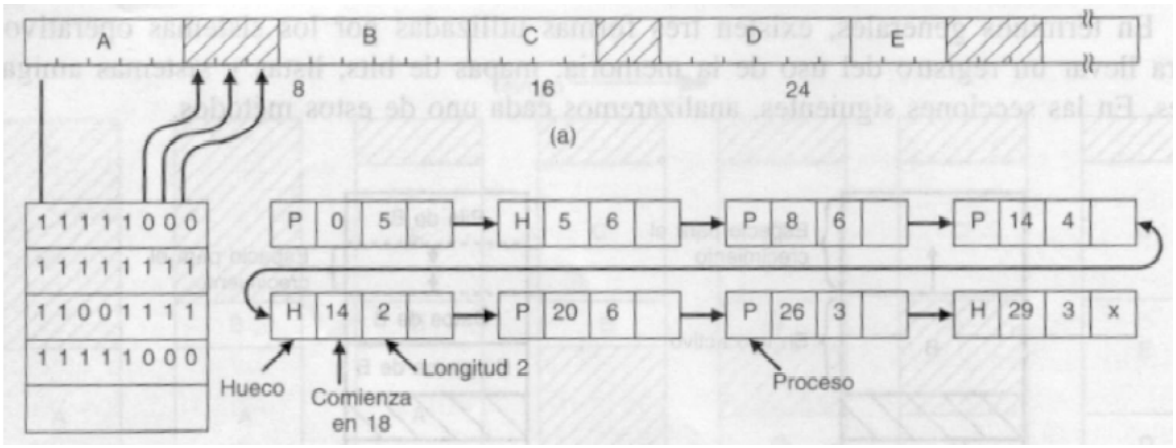


Figura 9: Administración de memoria con mapas de bits.
Fuente: Tanenbaum (2003: 199).

Podemos dividir la memoria en pequeñas unidades, y registrar en un mapa de bits las unidades ocupadas y desocupadas. Las unidades pueden ser de unas pocas palabras cada una, hasta de un par de KB. A mayor tamaño de las unidades, menor espacio ocupa el mapa de bits, pero puede haber mayor fragmentación interna. Desventaja: para encontrar hoyo de *n* unidades, hay que recorrer el mapa hasta encontrar *n* ceros seguidos (puede ser caro).

4.2.2. Administración de memoria con listas enlazadas

Otra forma de administrar la memoria es mediante **una lista ligada de segmentos**: estado (ocupado o en uso), dirección (de inicio), tamaño. Cuando un proceso termina o se pasa a disco, si quedan dos hoyos juntos, estos se funden en un solo segmento. Si la lista se mantiene ordenada por dirección, podemos usar uno de los siguientes algoritmos para escoger un hoyo donde poner un nuevo proceso.

- **first-fit:** Asignar el primer hoyo que sea suficientemente grande como para contener al proceso.
- **best-fit:** Asignar el menor hoyo en el que el proceso quepa.
- **worst-fit:** Asignar el mayor hoyo.

Cada vez que se asigna un hoyo a un proceso, a menos que quepa exactamente, se convierte en un segmento asignado y un hoyo más pequeño. Best-fit deja hoyos pequeños y worst-fit deja hoyos grandes (¿cuál es mejor?). Simulaciones han mostrado que first-fit y best-fit son mejores en términos de utilización de la memoria. First-fit es más rápido (no hay que revisar toda la lista). Se puede pensar en otras variantes.

En la imagen siguiente se muestra la forma de administración de memoria mediante una lista ligada.

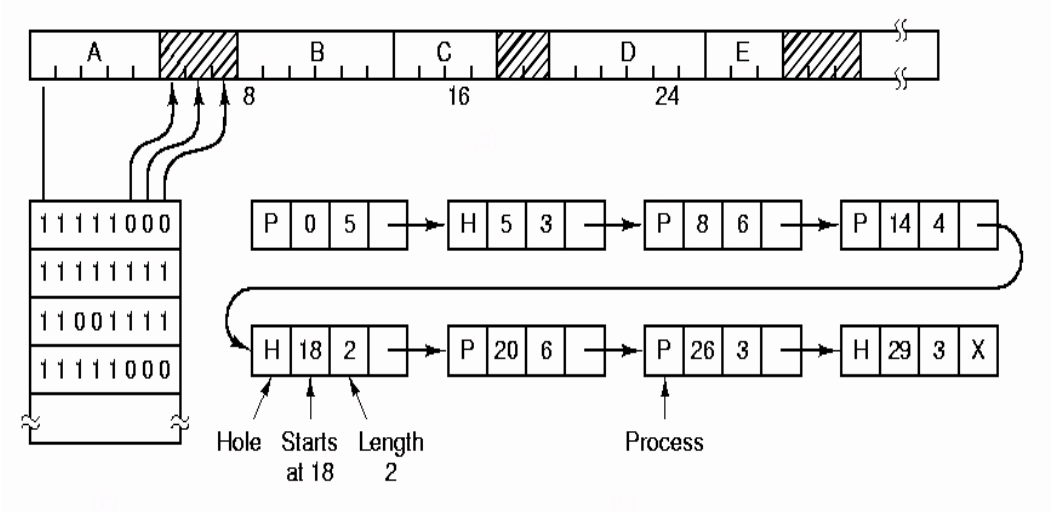


Figura 10: Administración de memoria con listas enlazadas.
Fuente: Trejo Ramírez (2002: 6).

4.3. Memoria virtual

¿Qué podemos hacer si un programa es demasiado grande para caber en la memoria disponible? Una posibilidad es usar **superposiciones** (*overlays*), como en MS-DOS: dividimos el programa en trozos independientes, y los vamos cargando de a uno, a medida que se necesiten. Por ejemplo, compilador de dos pasadas: cargar primero el código de la primera pasada, ejecutarlo, y después descartarlo para cargar el código de la segunda pasada. Las rutinas comunes y estructuras de datos compartidas entre las dos pasadas se mantienen en memoria permanentemente. El problema es que se agrega complejidad a la solución.

Mucho mejor sería poder extender la memoria de manera virtual, es decir, hacer que el proceso tenga la ilusión de que la memoria es mucho más grande que la memoria física (o que el trozo de memoria física que le corresponde, si tenemos multiprogramación). El sistema operativo se encarga de mantener en memoria física los trozos (páginas) que el proceso está usando, y el resto en disco. Ya que el disco es barato, podemos tener espacios de direccionamiento enormes.

4.3.1. Paginación

En sistemas operativos de computadoras, los sistemas de **paginación** de memoria dividen los programas en pequeñas partes o páginas. Del mismo modo, la memoria es dividida en trozos del mismo tamaño que las páginas, llamados **marcos de página**. De esta forma, la cantidad de memoria desperdiciada por un proceso es el final de su última página, lo que minimiza la fragmentación interna y evita la externa.

En un momento cualquiera, la memoria se encuentra ocupada con páginas de diferentes procesos, mientras que algunos marcos están disponibles para su uso. El sistema operativo mantiene una lista de estos últimos marcos, y una tabla por cada proceso, donde consta en qué marco se encuentra cada página del proceso. De esta forma, las páginas de un proceso pueden no estar contiguamente ubicadas en memoria, y pueden intercalarse con las páginas de otros procesos.

4.3.2. Tablas de páginas

En sistemas operativos de computadoras, los sistemas de **paginación** de memoria dividen los programas en pequeñas partes o páginas. Del mismo modo, la memoria es dividida en trozos del mismo tamaño que las páginas, llamados **marcos de página**. De esta forma, la cantidad de memoria desperdiciada por un proceso es el final de su última página, lo que minimiza la fragmentación interna y evita la externa.

En un momento cualquiera, la memoria se encuentra ocupada con páginas de diferentes procesos, mientras que algunos marcos están disponibles para su uso. El sistema operativo mantiene una lista de estos últimos marcos, y una tabla por cada proceso, donde consta en qué marco se encuentra cada página del proceso. De esta forma, las páginas de un proceso pueden no estar contiguamente ubicadas en memoria, y pueden intercalarse con las páginas de otros procesos.

En la tabla de páginas de un proceso, se encuentra la ubicación del marco que contiene a cada una de sus páginas. Las direcciones lógicas ahora se forman como un número de página y de un desplazamiento dentro de esa página (conocido comúnmente como ***offset***). El número de página es usado como un índice dentro de la tabla de páginas. Una vez obtenida la dirección del marco de memoria, se utiliza el desplazamiento para componer la dirección real o dirección física. Este proceso se realiza en una parte del computador específicamente diseñada para esta tarea, es decir, es un proceso *hardware* y no *software*.

Ahora que la memoria lógica puede ser mucho más grande que la física, se acentúa el problema del tamaño de la tabla de páginas. Con direcciones de 32 bits, puede alcanzar los 4GB. Con páginas de 4 KB, necesitaríamos hasta 1M entradas. Si cada entrada usa 32 bits, entonces la tabla podría pesar 4 MB. Además, cada proceso tiene su tabla, y esta tabla **debe** estar siempre en memoria física. Esto representa un problema serio.

Solución: tablas de varios niveles. El 386 usa dos niveles: una dirección lógica de 32 bits se divide en (p_1, p_2, d), de 10, 10 y 12 bits respectivamente.

p_1 es un índice a una tabla de tablas, metatabla, o tabla de primer nivel, que contiene $2^{10}=1024$ entradas de 32 bits, o sea, exactamente una página. La entrada correspondiente a p_1 en esta tabla indica el marco donde se encuentra otra tabla (de segundo nivel), donde hay que buscar indexando por p_2, el marco donde está la página que finalmente se necesita. A continuación, una representación de tabla de páginas:

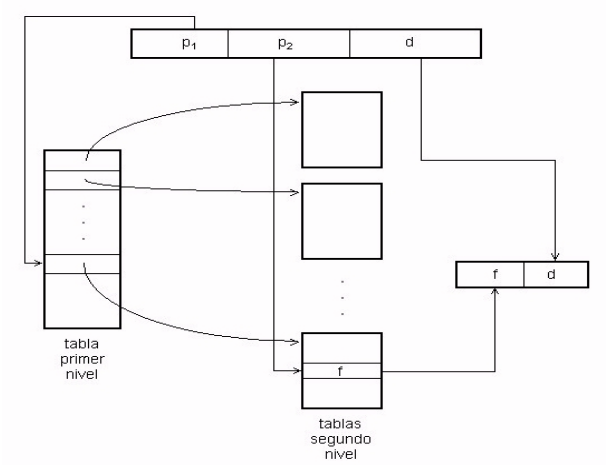


Figura 11: Representación de tabla de páginas.
Fuente: Gómez Jiménez, Enrique.

¿Por qué se ha resuelto el problema? La clave es que solo es indispensable mantener en memoria la tabla de primer nivel. Las otras se tratan como páginas **ordinarias**, es decir, pueden estar en disco. Así, al utilizar la propia memoria virtual para estas páginas, se mantienen en memoria física solo las más usadas. Sin embargo, se está agregando un acceso más a la memoria para convertir cada dirección lógica a una física, pero con un TLB con tasa de aciertos alta el impacto es mínimo.

Si las direcciones fueran de 64 bits, hay que usar más niveles. Otra posibilidad es usar **tablas de página invertidas**, en las que, en vez de haber una entrada por página virtual, hay una entrada por marco en la memoria física. Así, si tenemos 32 MB de memoria real y páginas de 4 KB, una tabla invertida necesita solo 8K entradas (independientemente de si las direcciones son de 32 o 64 bits). Cada entrada dice a qué página virtual de qué proceso corresponde el marco. Cuando el proceso *P* accesa la dirección lógica *l*, hay que buscar un par *(P,l)* en esta tabla. La posición de este par es el marco que hay que accesar. Problema obvio: la búsqueda en una tabla de 8192 entradas puede costar muchos accesos a memoria.

4.4. Algoritmos para reemplazo de páginas

Con el uso del método de paginación se puede llegar a saturar la memoria si se incrementa demasiado el nivel de multiprogramación. Por ejemplo, si se corren seis procesos, cada uno con un tamaño de diez páginas de las cuales en realidad solo utiliza cinco, se tiene un mayor uso del CPU y con marcos de sobra. Pero pudiera suceder que cada uno de esos procesos quiera usar las

diez páginas, lo que da como resultado una necesidad de 60 marcos, cuando solo hay 40 disponibles.

Esto provoca sobre-asignación. Y, mientras un proceso de usuario se está ejecutando, ocurre un fallo de página. El *hardware* se bloquea con el sistema operativo, el cual checa en sus tablas internas y se da cuenta de que es un fallo de página y no un acceso ilegal de memoria. El sistema operativo determina si la página está residiendo en disco, pero también determina que no hay marcos de memoria disponibles en la lista de marcos libres.

Al ocurrir el fallo de página, el sistema operativo debe elegir una página para retirarla de la memoria y usar el espacio para la página que se necesita para desbloquear el sistema y que el *hardware* pueda seguir trabajando. Si la página por eliminar de la memoria fue modificada, se debe volver a escribir al disco para mantener la información actualizada; de lo contrario, si la página no fue modificada no es necesario rescribir la información a disco y la página que se carga simplemente se escribe sobre la página por borrar en memoria. La siguiente figura muestra gráficamente un intercambio de páginas entre la memoria principal y el disco (memoria secundaria).

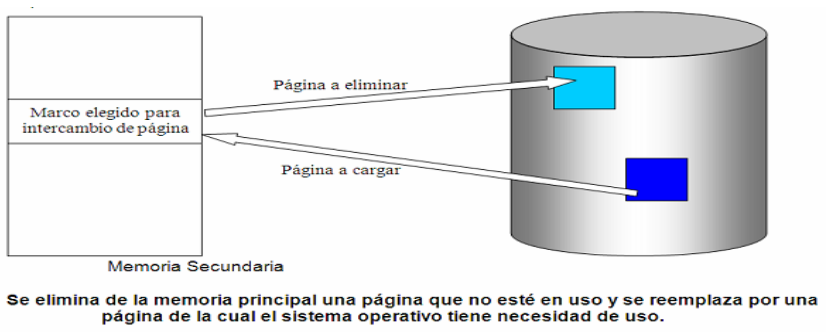


Figura 12: Intercambio de páginas entre la memoria principal y el disco.
Fuente: Trejo Ramírez (2002: 8).

Algoritmo de reemplazo de páginas óptimo

Este algoritmo debe tener el menor índice de fallos de página de todos los algoritmos. En teoría, este algoritmo debe reemplazar la página que no va a ser usada por el periodo más largo de tiempo.

Desafortunadamente, el algoritmo de reemplazo óptimo es fácil en teoría, pero prácticamente imposible de implementar, dado que requiere conocer a futuro las necesidades del sistema.

Tal algoritmo existe y ha sido llamado OPT o MIN. Pero se usa únicamente para estudios de comparaciones. Por ejemplo, puede resultar muy útil saber que aunque algún nuevo algoritmo no sea óptimo, está entre el 12,3% del óptimo y entre el 4,7% en promedio.

Algoritmo de reemplazo de páginas según el uso no tan reciente

Este algoritmo hace uso de los dos bits de estado que están asociados a cada página. Estos bits son: R, el cual se activa cuando se hace referencia (lectura / escritura) a la página asociada; y M, que se activa cuando la página asociada es modificada (escritura). Estos bits deben ser actualizados cada vez que se haga referencia a la memoria. Por esto es de suma importancia que sean activados por el *hardware*. Una vez activado el bit, permanece en ese estado hasta que el sistema operativo, mediante *software*, modifica su estado.

Estos bits pueden ser utilizados para desarrollar un algoritmo de reemplazo de forma que, cuando inicie el proceso, el sistema operativo asigne un valor de 0 a ambos bits en todas las páginas; que en cada interrupción de reloj, limpie el bit R para distinguir cuáles páginas tuvieron referencia y cuáles no.

Cuando ocurre un fallo de página, el sistema operativo revisa ambos bits en todas las páginas, y las clasifica de la siguiente manera:

- clase 0: La página no ha sido referenciada ni modificada.
- clase 1: La página no ha sido referenciada pero ha sido modificada.
- clase 2: La página ha sido referenciada pero no ha sido modificada.
- clase 3: La página ha sido referenciada y también modificada.

Una vez obtenida la clasificación, elimina una página de manera aleatoria de la primera clase no vacía con el número más pequeño. Esto ocurre porque para el algoritmo es mejor eliminar una página modificada sin referencias en al menos un intervalo de reloj, que una página en blanco de uso frecuente.

A pesar de que este algoritmo no es el óptimo, es fácil de implementar y de comprender, y con mucha frecuencia es el más adecuado.

Algoritmo de reemplazo *Primero en entrar, primero en salir* (*FIFO*)

El algoritmo más sencillo para reemplazo de páginas es el **FIFO (First In – First Out)**. Este algoritmo asocia a cada página el momento en que esta fue traída a memoria. Cuando una página debe ser reemplazada, se selecciona la más antigua.

No es estrictamente necesario registrar el momento de entrada de la página a memoria, sino que se puede crear una cola en la que se van agregando las páginas conforme van llegando a la memoria. Cuando se debe eliminar una página, se selecciona la que está al frente de la lista (o sea, la más antigua de la lista). Cuando llega una página nueva, se inserta en la parte trasera de la cola.

Al igual que el algoritmo aleatorio, este algoritmo es fácil de comprender y de programar. Sin embargo, su desempeño no siempre es del todo bueno. La página reemplazada puede ser un módulo de inicialización que fue usado hace mucho tiempo, y ya no se tiene necesidad de él. Por otro lado, puede contener una variable de uso muy frecuente que fue inicializada de manera temprana y está en uso constante. La imagen siguiente muestra este algoritmo.

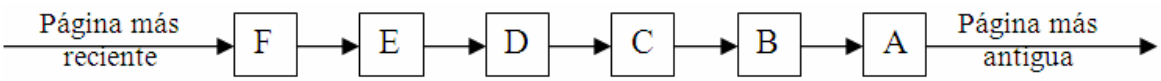


Figura 13: Reemplazo de páginas mediante el algoritmo FIFO.
Fuente: Trejo Ramírez (2002: 9).

Algoritmo de reemplazo de páginas de la segunda oportunidad

Este algoritmo es una modificación del FIFO. El algoritmo hace uso del bit de referencia de la página. Cuando una página ha sido seleccionada para reemplazo, se revisa el bit de referencia. Si tiene valor de 0, se procede a reemplazar la página. Si, por el contrario, el bit de referencia es 1, se le da a la página una segunda oportunidad. La siguiente figura muestra la funcionalidad de este algoritmo.

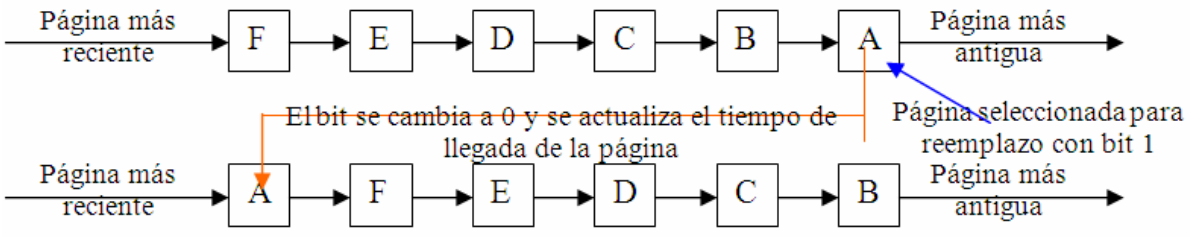


Figura 14: Algoritmo de la segunda oportunidad.
Fuente: Trejo Ramírez (2002: 9).

Algoritmo de reemplazo de páginas del reloj

Modificando el algoritmo de la segunda oportunidad (que a su vez es una modificación de FIFO), obtenemos el algoritmo aumentado de la segunda oportunidad o algoritmo del reloj. Se utiliza la misma clasificación del algoritmo de uso no tan reciente.

Este algoritmo organiza las páginas en una lista circular, como se muestra en la figura siguiente. Se usa un apuntador (o manecilla) que señala la página más antigua.

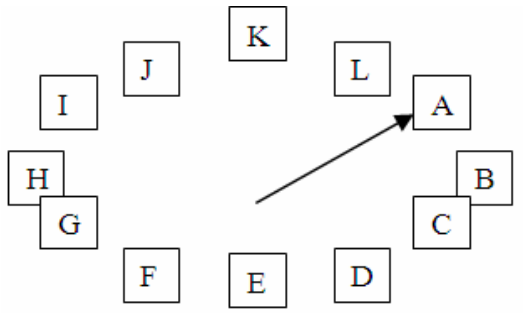


Figura 15: Algoritmo de reloj.
Fuente: Trejo Ramírez (2002: 10).

Algoritmo de reemplazo de páginas "la de menor uso reciente" (LRU)

El **algoritmo LRU** es una buena aproximación al óptimo. Se basa en la observación de que las páginas de uso frecuente en las últimas instrucciones, se utilizan con cierta probabilidad en las siguientes. De la misma manera, es probable que las páginas que no hayan sido utilizadas durante mucho tiempo permanezcan sin uso por bastante tiempo. Implementando el algoritmo con esta base, al ocurrir un fallo de página, se elimina la página que no haya sido utilizada durante el tiempo más grande. De ahí su denominación: **menor uso reciente (LRU - *Least Recent Use*)**.

A diferencia de los algoritmos anteriores, el LRU tiene un mejor rendimiento en cuanto al tiempo de aprovechamiento del CPU y del uso de la memoria. Sin embargo, el problema con este algoritmo es que su implementación es muy cara, ya que requiere de una asistencia considerable de *hardware*. Otro problema es el de determinar un orden para los marcos definido por el tiempo de menor uso.

Para este último hay dos posibles implementaciones:

- **Contadores:** En el caso más sencillo, se asocia en cada entrada tabla-página un campo de tiempo-de-uso, y se le agrega al CPU un reloj lógico o contador. Este reloj es incrementado en cada referencia de memoria. Siempre que se hace referencia a una página, el contenido del registro del reloj es copiado al campo de tiempo-de-uso en la tabla de páginas para esa página. De esta forma, siempre se dispone del "tiempo" de la última referencia a cada página. La página que se reemplaza es la del menor valor de tiempo. Este esquema requiere de una búsqueda en toda la tabla de páginas para encontrar la página LRU, y una escritura en memoria al campo de tiempo-de-uso en la tabla de páginas por cada acceso a memoria. Los tiempos también se deben mantener cuando las tablas de páginas son alteradas (debido a organización del CPU). Debe considerarse la posibilidad de sobrecarga en el reloj.

- **Pilas:** Otra aproximación para implementar el reemplazo LRU es la de tener una pila con los números de páginas. Siempre que se hace referencia a una página, se quita de la pila y se pone en la parte superior. De esta manera, la parte superior de la pila es la página de uso más reciente, y la de abajo es la LRU, tal como se muestra en la figura siguiente:

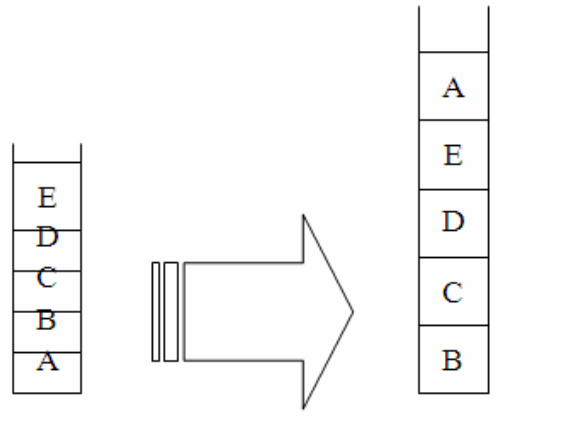


Figura 16: Uso de pilas en el algoritmo LRU.
Fuente: Trejo Ramírez (2002: 11).

4.5. Modelación de algoritmos de reemplazo de páginas

Cuando un proceso accesa una página inválida, hay que ir a buscar su contenido a disco y ponerlo en algún marco. El punto es determinar en qué marco, si no hay ninguno libre. Una posibilidad en tal caso es pasar a disco un proceso, con lo cual los marcos que ocupaba quedan libres.

Lo usual es usar algún **algoritmo de reemplazo de páginas** para seleccionar un marco **víctima** (por ejemplo, al azar). Para poder usar el marco víctima, primero hay que sacar la página contenida en ese marco (pasarla a disco). Sin embargo, si la página original no había sido modificada, la copia en disco es idéntica, así que se puede descartar esta operación de escritura. El sistema operativo puede saber si una página ha sido modificada examinando un bit asociado a la página (el *dirty bit* o bit de modificación) que el *hardware* se encarga de poner en 1 cada vez que se escribe en ella.

O sea, cada falta de página produce una o dos transferencias de páginas entre la memoria y el disco.

En síntesis:

- Previene la sobreasignación de memoria modificando la rutina de fallo de páginas para incluir el reemplazo de páginas.

- Se usa el **bit de modificado** (**suciedad**) para reducir la sobrecarga por transferencias de páginas (solo las páginas modificadas se escriben a disco).

- El reemplazo de páginas completa la separación entre la memoria lógica y la memoria física (se puede proporcionar una gran cantidad de memoria virtual sobre una menor memoria física).

En la siguiente imagen se muestra gráficamente la necesidad existente para el reemplazo de páginas:

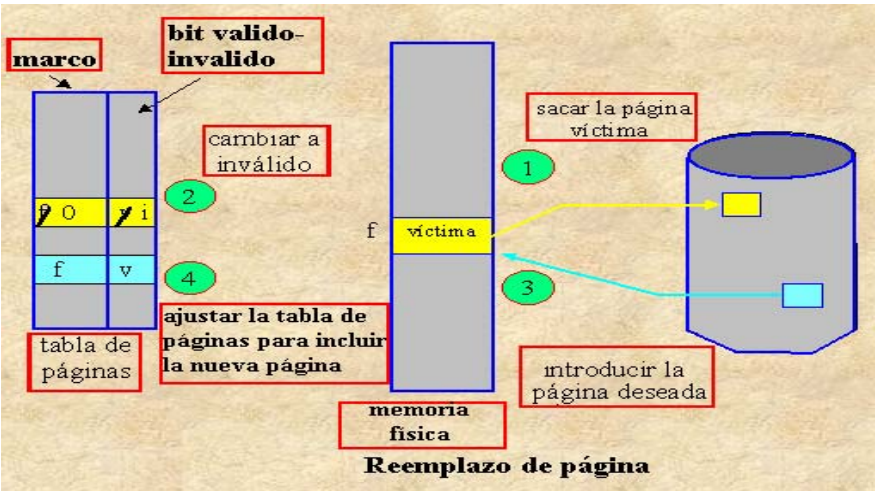


Figura 17: Reemplazo de página.
Fuente: Martínez y Hervás (1998: 1).

Anomalía de Belady

La **anomalía de Belady** es un efecto descubierto y demostrado en 1969 por el científico de la computación húngaro Laszlo Belady. Por este es posible tener más fallos de página al aumentar el número de marcos en la memoria física utilizando el método FIFO como algoritmo de reemplazo de páginas en sistemas de gestión de memoria virtual con paginación. Antes de esta fecha, se creía que incrementar el número de marcos físicos siempre llevaría a un descenso del número fallo de páginas o, en el peor de los casos, a mantenerlo. Así, pues, antes del descubrimiento de la anomalía de Belady, el algoritmo FIFO era aceptable.

El siguiente es un ejemplo de la anomalía de Belady. Utilizando tres marcos ocurren 9 fallos de página. Aumentando a cuatro marcos, obtenemos 10 fallos de página. Los fallos de página están en rojo y subrayado. El contenido de la memoria principal en cada momento está debajo de cada nueva petición de página.

Peticiones de página	**3**	**2**	**1**	**0**	**3**	**2**	**4**	**3**	**2**	**1**	**0**	**4**
Página más nueva	3	2	1	0	3	2	4	4	4	1	0	0
		3	2	1	0	3	2	2	2	4	1	1
Página más antigua			3	2	1	0	3	3	3	2	4	4

Peticiones de página	**3**	**2**	**1**	**0**	**3**	**2**	**4**	**3**	**2**	**1**	**0**	**4**
Página más nueva	3	2	1	0	0	0	4	3	2	1	0	4
		3	2	1	1	1	0	4	3	2	1	0

	3 2 2 2 1 0 4 3 2 1
Página más antigua	3 3 3 2 1 0 4 3 2

(rojo indica fallo de página)

Algoritmos de pila

Un **algoritmo de pila** es aquel para el cual se puede demostrar que el conjunto de páginas en memoria para n marcos es siempre un subconjunto del conjunto de las páginas que estarían en memoria con n + 1 marcos.

Para implementar el reemplazo LRU es necesario tener una pila con los números de páginas. Siempre que se hace referencia a una página, se quita de la pila y se pone en la parte superior. De esta manera, la parte superior de la pila es la página de uso más reciente y la de abajo es la LRU, tal y como se muestra en la figura siguiente:

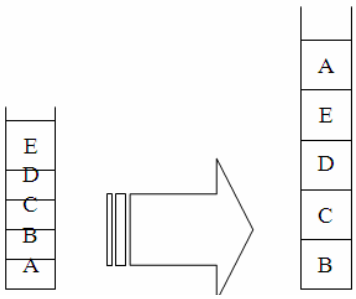


Figura 18: Uso de pilas en el algoritmo LRU.
Fuente: Trejo Ramírez (2002: 11).

4.6. Aspectos de diseño de los sistemas de paginación

Hay algunos factores que deben considerarse en el momento de **diseñar la paginación de un sistema operativo**. Entre ellos:

a) **Políticas de asignación local y global**, que consisten en considerar cómo debe repartirse la memoria entre los procesos ejecutables que compiten por ella.
b) **Control de carga**, que consiste en controlar que el conjunto de trabajos de todos los procesos no excedan la capacidad de la memoria creando hiperpaginación.
c) **Tamaño de página**, que consiste en determinar el tamaño de página requerido para el manejo y almacenamiento de los datos.

d) Espacio de instrucciones y de datos separados. Casi todas las computadoras poseen un único espacio de direcciones donde se almacenan datos y programas.
e) Otros factores

4.7. Aspectos de implementación

Para la **implementación de la paginación del sistema operativo**, deben considerarse factores como los siguientes:

a) Cómo el sistema operativo debe intervenir en la paginación.
b) Cómo actuar en caso de falla de la paginación.
c) Cómo fijar las páginas en la memoria del computador.
d) Otros factores.

4.8. Segmentación

La **segmentación de memoria** es un esquema de manejo de memoria, mediante el cual la estructura del programa refleja su división lógica, llevándose a cabo una agrupación lógica de la información en bloques de tamaño variable denominados **segmentos**.

Cada segmento contiene información lógica del programa: subrutina, arreglo, etc. Luego, cada espacio de direcciones de programa consiste de una colección de segmentos, que generalmente reflejan la división lógica del programa.

Algunos factores para la implementación de la segmentación deben ser considerados. Por ejemplo, cómo implementar la segmentación pura o por paginación.

Capítulo 5: Principios de *Hardware* de E/S

5.1. Principios de *hardware* de E/S

Dispositivos de E/S

Se pueden clasificar en dos grandes categorías:

- **Dispositivos de bloque**
- **Dispositivos de carácter**

Las principales características de los **dispositivos de bloque** son:

- La información se almacena en bloques de tamaño fijo.
- Cada bloque tiene su propia dirección.
- Los tamaños más comunes de los bloques van desde los 128 bytes hasta los 1.024 bytes.
- Se puede leer o escribir en un bloque de forma independiente de los demás, en cualquier momento.
- Un ejemplo típico de dispositivos de bloque son los discos.

Las principales características de los **dispositivos de carácter** son:

- La información se transfiere como un flujo de caracteres, sin sujetarse a una estructura de bloques.
- No se pueden utilizar direcciones.
- No tienen una operación de búsqueda.
- Unos ejemplos típicos de dispositivos de carácter son las impresoras de línea, terminales, interfaces de una red, ratones, etc.

Algunos dispositivos no se ajustan a este esquema de clasificación; por ejemplo, los relojes, que no tienen direcciones por medio de bloques, y no generan o aceptan flujos de caracteres.

El **sistema de archivos** solo trabaja con **dispositivos de bloque abstractos**, por lo que encarga la parte **dependiente del dispositivo** a un *software* de menor nivel, el ***software* manejador del dispositivo***.*

Controladores de E/S

Las unidades de E/S generalmente constan de:

- Un **componente mecánico**
- Un componente electrónico, el **controlador del dispositivo** o **adaptador**

Muchos controladores pueden manejar más de un dispositivo.

El SO generalmente trabaja con el controlador y no con el dispositivo.

Los modelos más frecuentes de **comunicación entre el CPU y los controladores** son:

  - Para la mayoría de las micro y mini computadoras: Modelo de **bus del sistema**.
  - Para la mayoría de los ***mainframes***: Modelo de varios buses y computadoras especializadas en E/S llamadas *canales de E/S.*

La **interfaz entre el controlador y el dispositivo** es con frecuencia de muy bajo nivel:

- La comunicación es mediante un **flujo de bits en serie** que:
  - Comienza con un preámbulo.
  - Sigue con una serie de bits (de un sector de disco, por ejemplo).
  - Concluye con una suma para verificación o un código corrector de errores.

- El preámbulo:
  - Se escribe al dar formato al disco.
  - Contiene el número de cilindro y sector, el tamaño de sector y otros datos similares.

El controlador debe:

- Convertir el flujo de bits en serie en un bloque de bytes.
- Efectuar cualquier corrección de errores necesaria.
- Copiar el bloque en la memoria principal.

Cada controlador posee registros que utiliza para comunicarse con el CPU:

- Pueden ser parte del espacio normal de direcciones de la memoria: *E/S mapeada a memoria.*
- Pueden utilizar un espacio de direcciones especial para la E/S, asignando a cada controlador una parte de él.

EL SO realiza la E/S al escribir comandos en los registros de los controladores; los parámetros de los comandos también se cargan en los registros de los controladores.

Al *aceptar el comando*, el CPU puede dejar al controlador y dedicarse a otro trabajo.

Al *terminar el comando*, el controlador *provoca una interrupción* para permitir que el *SO*:

- Obtenga el control del CPU.
- Verifique los resultados de la operación.

El CPU **obtiene los resultados** y el estado del dispositivo al leer uno o más bytes de información de los registros del controlador.

Acceso directo a memoria DMA

Muchos controladores, especialmente los correspondientes a dispositivos de bloque, permiten el DMA.

Si se lee el disco **sin** *DMA:*

- El controlador lee en serie el bloque (uno o más sectores) de la unidad:
  - La lectura es bit por bit.
  - Los bits del bloque se graban en el buffer interno del controlador.
- Se calcula la suma de verificación para corroborar que no existen errores de lectura.
- El controlador provoca una interrupción.
- El SO lee el bloque del disco por medio del buffer del controlador:
  - La lectura es por byte o palabra a la vez.
  - En cada iteración de este ciclo se lee un byte o una palabra del registro del controlador y se almacena en memoria.
- **Se desperdicia tiempo del CPU.**

DMA se ideó para liberar al CPU de este trabajo de bajo nivel.

EL CPU le proporciona al controlador:

- La dirección del bloque en el disco
- La dirección en memoria a donde debe ir el bloque
- El número de bytes por transferir

Luego de que el controlador leyó todo el bloque del dispositivo a su buffer, y de que corroboró la suma de verificación,

- Copia el primer byte o palabra a la memoria principal.
- Lo hace en la dirección especificada por medio de la *dirección de memoria de DMA*.
- Incrementa la *dirección DMA* y decrementa el *contador DMA* en el número de bytes que acaba de transferir.
- Se repite este proceso hasta que el contador se anula, y por lo tanto el controlador provoca una interrupción.
- Al iniciar su ejecución el SO luego de la interrupción provocada, no debe copiar el bloque en la memoria, porque ya se encuentra ahí. Vea la imagen siguiente:

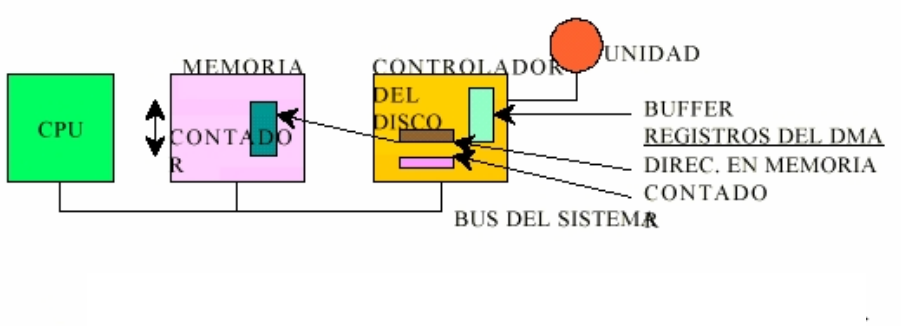


Figura 19: Un controlador realiza completamente una transferencia DMA.
Fuente: Tanenbaum (2003: 277).

El controlador necesita un buffer interno, ya que, una vez iniciada la transferencia del disco, ocurre lo siguiente:

- Los bits siguen llegando del disco constantemente.
- No interesa si el controlador está listo o no para recibirlos.
- Si el controlador intentara escribir los datos en la memoria directamente:
  - Tendría que recurrir al bus del sistema para c / u de las palabras (o bytes) transferidas.
  - El bus podría estar ocupado por otro dispositivo y el controlador debería esperar.

- Si la siguiente palabra llegara antes de que la anterior hubiera sido almacenada, el controlador la tendría que almacenar en alguna parte.

Si el bloque se guarda en un **buffer interno***:*

- El bus no se necesita sino hasta que el DMA comienza.
- La transferencia DMA a la memoria ya no es un aspecto crítico del tiempo.

Los controladores simples no pueden atender la E/S simultánea:

- Mientras transfieren a la memoria, el sector que pasa debajo de la cabeza del disco se pierde, es decir, que el bloque siguiente al recién leído se pierde.
- La lectura de una pista completa se hará en dos rotaciones completas, una para los bloques pares y otra para los impares.
- Si el tiempo necesario para una transferencia de un bloque del controlador a la memoria por medio del bus es mayor que el tiempo necesario para leer un bloque del disco:
  - Sería necesario leer un bloque y luego saltar dos o más bloques.
  - El salto de bloques:
    - Se ejecuta para darle tiempo al controlador para la transferencia de los datos a la memoria.
    - Se llama separación.
    - Al formatear el disco, los bloques se numeran tomando en cuenta el factor de separación.
    - Esto permite al SO:
      - Leer los bloques con numeración consecutiva.
      - Conservar la máxima velocidad posible del *hardware*.

5.2. Principios de *software* de E/S

La idea básica es organizar el *software* como una **serie de capas** donde:

- Las **capas inferiores** se encarguen de ocultar las peculiaridades del *hardware* a las capas superiores.
- Las **capas superiores** deben presentar una interfaz agradable, limpia y regular a los usuarios.

Metas del *software* de E/S

Un concepto clave es ***la independencia del dispositivo****:*

- Debe ser posible escribir programas que se puedan utilizar con **archivos en distintos dispositivos**, sin tener que modificar los programas para cada tipo de dispositivo.
- El problema debe ser resuelto por el SO.

El objetivo de lograr **nombres uniformes** está muy relacionado con el de independencia del dispositivo.

Todos los archivos y dispositivos adquieren direcciones de la misma forma, es decir, mediante el **nombre de su ruta de acceso***.*

Otro aspecto importante del *software* es el *manejo de errores de E/S:*

1. Generalmente **los errores deben manejarse lo más cerca posible del *hardware****.***
2. Solo si los niveles inferiores no pueden resolver el problema, se informa a los niveles superiores.
3. Generalmente, la recuperación se puede hacer en un nivel inferior y de forma transparente.

Otro aspecto clave son las **transferencias síncronas** (por bloques) o **asíncronas** (controlada por interruptores):

- La mayoría de la E/S es **asíncrona***:* EL CPU inicia la transferencia y realiza otras tareas hasta una interrupción.
- La programación es más fácil si la E/S es **síncrona** (por bloques): El programa se suspende automáticamente hasta que los datos estén disponibles en el buffer.

El SO se encarga de hacer que operaciones controladas por interruptores parezcan del tipo de bloques para el usuario.

También el SO debe administrar los dispositivos compartidos (ejemplo: discos) y los de uso exclusivo (ejemplo: impresoras).

Generalmente, el *software* de E/S se estructura en capas:

- Manejadores de interrupciones
- Directivas de dispositivos
- *Software* de SO independiente de los dispositivos
- *Software* a nivel usuario

E/S programada

Los datos se intercambian entre el CPU y el módulo de E/S. El CPU ejecuta un programa que controla directamente la operación de E/S, incluyendo la comprobación del estado del dispositivo, el envío de la orden de lectura o escritura y la transferencia del dato. Cuando el CPU envía la orden, debe esperar hasta que la operación de E/S concluya. Si el CPU es más rápido, este estará ocioso. El CPU es el responsable de comprobar periódicamente el estado del módulo de E/S hasta que encuentre que la operación ha finalizado.

Normalmente habrá muchos dispositivos de E/S conectados al sistema a través de los módulos de E/S. Cada dispositivo tiene asociado un identificador o dirección. Cuando el CPU envía una orden de E/S, la orden contiene la dirección del dispositivo deseado.

El problema con la E/S programada es que el CPU tiene que esperar un tiempo considerable para que el módulo de E/S concerniente esté listo para la recepción o transmisión de datos. Y el hecho de que el CPU deba interrogar repetidamente por el status del módulo de E/S hace que el nivel de performance de todo el sistema sea severamente degradado.

E/S controlada por interrupciones

Permite que el CPU esté ocupado en alguna otra actividad mientras que se realiza la operación de E/S, pues se enterará de que dicha operación se ha completado cuando se produzca una interrupción.

Las interrupciones son un mecanismo que permite sincronizar el CPU con los sucesos externos, y por lo tanto solapar una multitud de operaciones de E/S.

Cuando la interrupción desde un módulo de E/S ocurre, el CPU salva el contexto de ejecución del programa corriente y procesa la interrupción. La E/S por interrupción es más eficiente que la E/S programada porque elimina la espera innecesaria del CPU por la E/S. Sin embargo, todavía se consume una gran cantidad de tiempo del procesador. Este es el responsable de la transferencia de datos.

La ocurrencia de una interrupción dispara un número de eventos, tanto en el procesador en *hardware* como en el *software*. Cuando un dispositivo de E/S completa una operación de E/S, ocurre una secuencia de eventos en *hardware*.

Una alternativa es que el CPU, tras enviar una orden de E/S, continúe realizando algún trabajo útil. El módulo de E/S interrumpirá al CPU para solicitar su servicio cuando esté preparado para intercambiar datos. El CPU ejecuta la transferencia de datos, y después continúa con el procesamiento previo.

Se pueden distinguir dos tipos: E/S sincrónica y E/S asincrónica

- **E/S sincrónica:** Cuando la operación de E/S finaliza, el control es retornado al proceso que la generó. La espera por E/S se lleva a cabo por medio de una instrucción *wait* que coloca al CPU en un estado ocioso hasta que ocurre otra interrupción. Aquellas máquinas que no tienen esta instrucción utilizan un *loop*. Este *loop* continúa hasta que ocurre una interrupción transfiriendo el control a otra parte del sistema operativo. Solo se atiende una solicitud de E/S por vez. El sistema operativo conoce exactamente qué dispositivo está interrumpiendo. Esta alternativa excluye el procesamiento simultáneo de E/S.

- **E/S asincrónica:** Retorna al programa usuario sin esperar que la operación de E/S finalice. Se necesita una llamada al sistema que le permita al usuario esperar por la finalización de E/S (si es requerido). También es necesario llevar un control de las distintas solicitudes de E/S. Para ello, el sistema operativo utiliza una tabla que contiene una entrada por cada dispositivo de E/S (Tabla de Estado de Dispositivos). La ventaja de este tipo de E/S es el incremento de la eficiencia del sistema. Mientras se lleva a cabo E/S, el CPU puede ser usado para procesar o para planificar otras E/S. Como la E/S puede ser bastante lenta comparada con la velocidad del CPU, el sistema hace un mejor uso de los recursos.

Inicio de la operación de E/S

- Para iniciar una operación de E/S, el CPU actualiza los registros necesarios en el módulo de E/S.
- El módulo de E/S examina el contenido de estos registros para determinar el tipo de acción que debe ser llevada a cabo. Por ejemplo, si encuentra un requerimiento de lectura, el módulo de E/S empezará a transferir data desde el dispositivo a los buffers locales. Una vez terminada la transferencia, el módulo informa al CPU que la operación ha terminado por medio de una interrupción.

Procesamiento de la interrupción:

Cuando un dispositivo de E/S termina una operación de E/S se produce la siguiente secuencia de eventos:

- El dispositivo envía una señal de interrupción al procesador.
- El procesador termina la ejecución de la instrucción en curso antes de responder a la interrupción.
- El procesador comprueba si hay alguna interrupción. Si hay alguna, envía una señal de reconocimiento al dispositivo que la originó.
- El procesador debe prepararse para transferir el control a la rutina de interrupción. Debe guardar la información necesaria para continuar con el proceso en curso en el punto en que se interrumpió. Guarda en la pila del sistema el contenido de los registros, etc.
- El procesador carga en el PC la dirección de inicio del programa de gestión o servicio de interrupción solicitada.
- Una vez modificado el PC, el procesador continúa con el ciclo de instrucción siguiente. Es decir, se transfiere el control a la rutina servidora de la interrupción.
- Cuando finaliza el servicio de la interrupción, se restauran los valores de los registros.

E/S con DMA

Tanto en la E/S programada o por interrupciones el CPU es responsable de extraer los datos de la memoria central para el output o almacenar los datos en la memoria central para el input. La alternativa a esto se conoce como Acceso Directo a Memoria (DMA). En este modo, el módulo de E/S y la memoria central intercambian datos directamente, sin involucrar al CPU.

El módulo de DMA es capaz de imitar al CPU y de relevarlo en el control del sistema para transferir los datos con la memoria por el bus de sistema. Cuando el CPU desea leer o escribir un bloque de datos, emite un comando al módulo de DMA, enviándole la siguiente información: si el pedido es una escritura o una lectura, la dirección del dispositivo de E/S involucrado, la ubicación en memoria desde dónde empezar a leer o escribir y el número de palabras escritas o leídas.

El CPU continúa con otro trabajo. Ha delegado esta operación de E/S al módulo de DMA. Cuando la transferencia termina el módulo DMA manda una señal de interrupción al procesador. El CPU solo opera al comienzo y al final de la transferencia.

El módulo de DMA utiliza el bus solo cuando el CPU no lo necesita o debe forzarla a que suspenda temporalmente su operación. Esta última operación se la conoce como **robo de ciclo** (DMA roba un ciclo del bus). Esto no es una interrupción ya que el CPU no tiene que guardar el contexto.

5.3. Capas del *software* de E/S

Podemos resumir las capas del *software* en cuanto E/S como sigue:

1. *Software* en espacio de usuario:
    - Realizan la llamada de E/S
    - Dan formato a la E/S
    - Gestionan la cola de impresión
2. *Software* de E/S independiente del dispositivo:
    - Asigna nombres, protección, bloques, búferes y drivers
3. Controladores de dispositivos:
    - Coloca órdenes en los registros del dispositivo
    - Comprueba el estado del dispositivo
4. Manejador de interrupciones:
    - "Despierta" al driver cuando se ha completado la operación de E/S
5. *Hardware*:
    - Realiza la operación de E/S

5.4. Discos

Hardware de disco

Un disco se organiza en cilindros, cada uno con tantas pistas como cabezas de lectura/grabación haya. Las pistas se dividen en sectores, cada uno con el mismo número de bytes.

Formateo de disco

El sistema operativo podrá acceder a la información que se encuentra en el disco duro, solamente que éste esté formateado de bajo nivel, particionado y de alto nivel.

El formateo de bajo nivel es la creación de las estructuras físicas, es decir, pistas sectores, etc. Este formateo lo hace el fabricante, o sea que, cuando nosotros compramos el disco, ya trae este formateo consigo.

El formateo de alto nivel es el que construye las estructuras lógicas del disco y almacena algunos archivos indispensables para el funcionamiento del sistema operativo, y cada sistema operativo realiza este formateo de alto nivel a su manera. Por eso hay distintos sistemas de archivos, y no todos son compatibles entre sí. Cuando un sistema operativo hace el formateo de alto nivel crea el *Master Boot Record* y *File Allocation Table*.

Sobre las particiones. Cuando tenemos una sola partición en nuestro disco se dice que es una partición primaria; para crear nuevas secciones o unidades en el disco duro accesible al sistema operativo, necesitamos una partición extendida. Por ejemplo, si tenemos un disco con capacidad de 20 GB y queremos dividirlo en tres partes una de 12 GB, otra de una de 2 GB y otra de 6 GB, debemos crear una partición primaria de 12 GB y luego una extendida de 8 GB. En esta última tendremos que crear dos particiones lógicas una de 2 GB y otra de 6 GB.

Los sistemas operativos Windows cuentan con su propia herramienta para crear particiones. Una de ellas es FDISK. Lo malo en FDISK es que no se pueden manipular particiones que contengan datos. Primero tendremos que respaldar los datos y borrarlos, para que nos permita hacer la partición. Se denomina **partición activa** a la partición que contiene el sistema operativo.

También podemos utilizar programas comerciales que no tienen la limitante de FDISK, como por ejemplo *Partition Magic*. Con él podemos crear dos particiones de nuestro disco no importando que el disco duro tenga datos en él, podemos fusionar dos particiones en una sola, o podemos modificar el tamaño de particiones existentes.

Algoritmos para calendarizar el brazo del disco

El tiempo requerido para leer o escribir un bloque está determinado por 3 factores:

1. Tiempo de búsqueda
2. Tiempo de rotación
3. Tiempo de transferencia

Los procesos de los usuarios pueden realizar peticiones de lectura/escritura en un disco más rápidamente de lo que pueden ser atendidas, lo que genera una cola de peticiones. Algunos de los algoritmos que se utilizan para gestionar esta cola son:

a) FCFS (*First Come, First Served*)

Primera en llegar, primera en atenderse.

El driver de disco acepta las demandas de una en una, y las atiende según el orden de llegada, con lo que no se optimiza el tiempo de búsqueda.

Es un algoritmo justo. Su comportamiento es aceptable cuando la carga de peticiones es "ligera" (no se realizan muchas), pero al aumentar la carga se incrementan los tiempos de respuesta. Su funcionalidad se muestra en la figura siguiente:

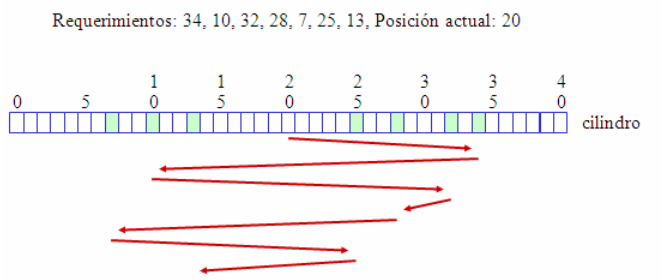


Figura 20: Algoritmo FIFO.
Fuente: s. a. *Dispositivos de entrada / salida* (s. f.:38).

b) SSF (*Shortest Seek First*)

Atiende primero a la demanda más próxima.

El driver de disco mantiene una tabla indexada por número de cilindro con todas las demandas pendientes para cada cilindro en una lista. De esta forma, cuando el driver de disco termina de atender todas las demandas de un cilindro, pasa a atender las demandas del cilindro más próximo para minimizar el tiempo de búsqueda.

Consecuencias:

- Mejora las tasas de capacidad de ejecución.
- Baja los tiempos de respuesta para cargas moderadas.
- Discrimina los cilindros externos para cargas pesadas. Si se reciben muchas demandas en el centro del disco, las demandas de los cilindros de los extremos tendrán que esperar mucho tiempo. Posible solución: mientras se atiende una cola de demandas, no se introducen más demandas en la misma (sería una variante del FCFS pero considerando grupos de demandas).

En la figura siguiente se muestra la funcionalidad de este algoritmo:

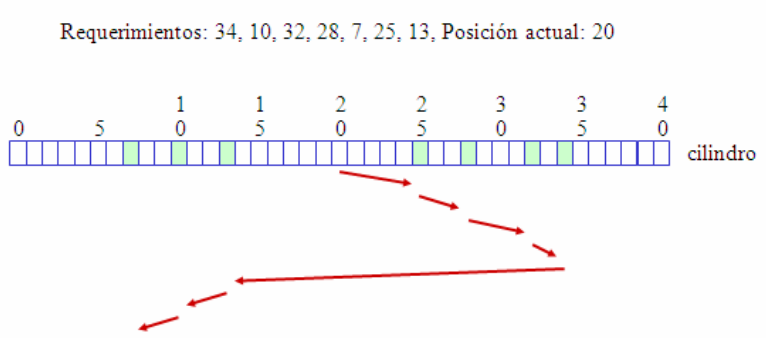


Figura 21: Algoritmo del menor tiempo de búsqueda.
Fuente: s. a. *Dispositivos de entrada / salida* (s. f.: 39).

c) Algoritmo del ascensor (SCAN)

El brazo de lectura/grabación del disco solo se desplaza en un sentido hasta que no haya más demandas (en ese sentido), en cuyo caso cambia de sentido (un bit indica el sentido de desplazamiento). De esta forma, cuando el driver de disco termina de atender una demanda, comprueba el bit, y si es "UP" (sentido ascendente) atiende a la siguiente demanda en sentido ascendente (si la hubiera). Si no existen más demandas en ese sentido, el bit se pone a "DOWN" (sentido descendente), y el brazo se desplaza en sentido contrario en busca de la siguiente demanda.

En la figura siguiente se muestra la funcionalidad de este algoritmo:

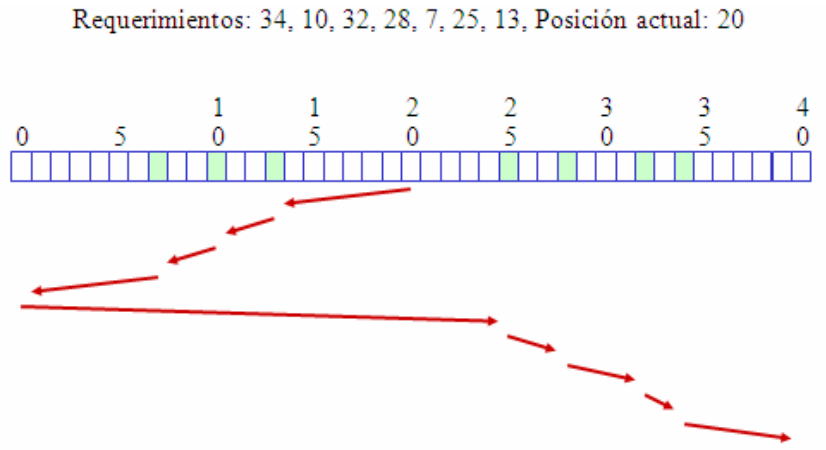


Figura 22: Algoritmo del elevador (SCAN).
Fuente: s. a. *Dispositivos de entrada / salida* (s. f.: 40)

Manejo de errores en discos

Algunos de los **errores más comunes** en discos son:

- **Error de programación:**
  Ejemplo: Solicitar un sector no existente.

- **Error temporal en la suma de verificación:**
  Ejemplo: Provocado por polvo en la cabeza.
- **Error permanente en la suma de verificación:**
  Ejemplo: Un bloque del disco dañado físicamente.
- **Error de búsqueda:**
  Ejemplo: El brazo se envía al cilindro 6 pero va al 7.
- **Error del controlador:**
  Ejemplo: El controlador no acepta los comandos.

El manejador del disco debe controlar los errores de la mejor manera posible.

La mayoría de los controladores:

- Verifican los parámetros que se les proporcionan.
- Informan si no son válidos.

Respecto de los **errores temporales** en la **suma de verificación***:*

- Generalmente se eliminan al repetir la operación.
- Si persisten, el bloque debe ser marcado como un bloque defectuoso, para que el *software* lo evite.

Otra posibilidad es que **controladores "inteligentes"** reserven cierta cantidad de pistas:

- Serán asignadas en reemplazo de pistas defectuosas.
- Una tabla asocia las **pistas defectuosas** con las **pistas de repuesto***:*
  - Está alojada en la memoria interna del controlador y en el disco.
  - La sustitución es transparente para el manejador.
  - Puede afectarse el desempeño de los algoritmos de búsqueda, como el del elevador, ya que el controlador utiliza pistas físicamente distintas de las solicitadas.

5.5. Relojes

Los relojes o cronómetros son esenciales para la operación de sistemas de tiempo compartido.

- Registran la hora del día.
- Evitan que un proceso monopolice el CPU.

El *software* para reloj toma generalmente la forma de un manejador de dispositivo, aunque no es un dispositivo de bloque, ni de caracter.

Hardware del reloj:

Dos tipos de relojes:

a) Relojes conectados a una línea de alimentación eléctrica de 110 a 220 voltios y causan una interrupción en cada ciclo de voltaje, a 50 o 60Hz. Son poco comunes
b) Relojes programables, formados por tres componentes: un oscilador de cristal de cuarzo, un contador y un registro. Generan una señal periódica de gran precisión.

Modos de operación habituales en un reloj programable:

- **Modo de disparo único**: Al poner en marcha el reloj, se copia el valor del registro en el contador. En cada pulso el contador disminuye. Cuando el contador llega a 0, se produce una interrupción, y se detiene hasta que el *software* lo vuelve a iniciar de forma explícita.
- **Modo de onda cuadrada**: Cuando el contador llega a 0 y se provoca una interrupción, el registro se copia de forma automática en el contador, y todo el proceso se repite.

Estas interrupciones periódicas se llaman **tics de reloj.**

Funciones principales:

- Registran la hora del día.
- Evitan que un proceso monopolice el CPU.

***Software* del reloj:**

Las principales funciones del *software* manejador del reloj son:

- Mantener la hora del día o tiempo real.
- Evitar que los procesos se ejecuten durante más tiempo del permitido.
- Mantener un registro del uso del CPU.
- Controlar llamadas al sistema tipo "*alarm*" por parte de los procesos del usuario.
- Proporcionar cronómetros guardianes de partes del propio sistema.
- Realizar resúmenes, monitoreo y recolección de estadísticas.

El *software* manejador del reloj puede tener que simular varios relojes virtuales con un único reloj físico.

5.6. Terminales orientadas a caracteres

Las terminales tienen gran número de formas distintas.

- El **manejador de la terminal** debe ocultar estas diferencias.
- La parte **independiente del dispositivo** en el SO y los **programas del usuario** no se tienen que reescribir para cada tipo de terminal.

Desde el punto de vista del SO, **se las puede clasificar en***:*

- **interfaz RS-232:**
  - Hardcopy (terminales de impresión)
  - TTY "de vidrio" (terminales de video)
  - Inteligente (computadoras con CPU y memoria)
- **interfaz mapeada a memoria:**
  - Orientada a caracteres
  - Orientada a bits

Las **terminales RS-232** poseen un teclado y un monitor que se comunican mediante una **interfaz serial**, un bit a la vez; las conversiones de bits a bytes y viceversa las efectúan los chips UART (transmisores - receptores asíncronos universales).

La figura siguiente muestra cómo trabaja una terminal orientada a caracteres (RS-232) y el *hardware* requerido para su funcionamiento:

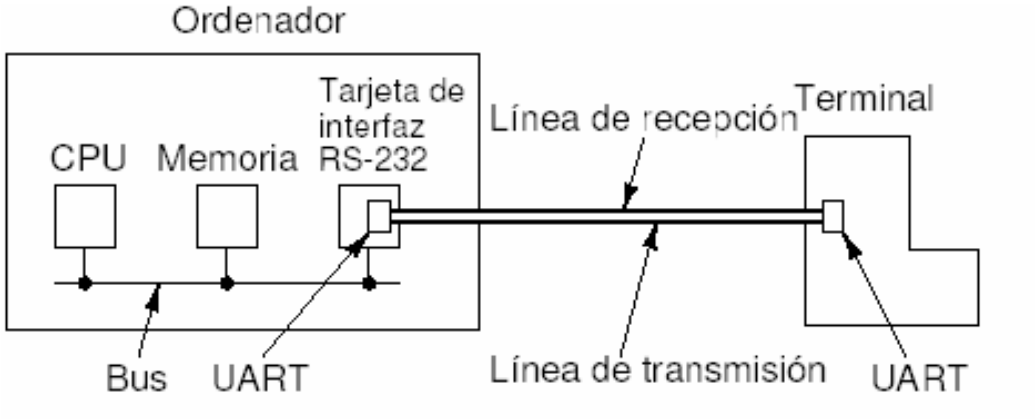


Figura 23: Terminal RS232.
Fuente: s. a. *Dispositivos de entrada / salida* (s. f.: 47).

Las **terminales mapeadas a memoria***:*

- No se comunican mediante una línea serial.
- Poseen una interfaz mediante una memoria especial llamada *video RAM:*
    - Forma parte del espacio de direcciones de la computadora.
    - El CPU se dirige a ella como al resto de la memoria.
    - En la tarjeta de *video RAM* hay un chip llamado **controlador de video**:
        - Extrae bytes del video RAM y genera la señal de video utilizada para manejar la pantalla.
        - El monitor genera un rayo de electrones que recorre la pantalla pintando líneas.
        - Cada línea está constituida por un cierto número de puntos o pixeles.
        - La señal del controlador de video modula el rayo de electrones y determina si un pixel debe estar o no iluminado.
        - Los monitores de color poseen tres rayos (rojo, verde y azul) que se modulan independientemente.

La siguiente imagen muestra la arquitectura del *hardware* de una **terminal con mapa de memoria**.

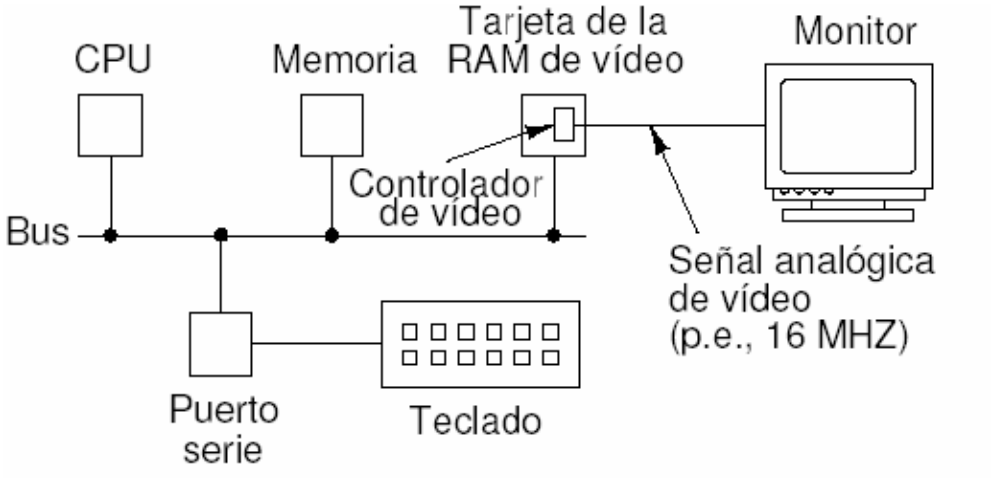


Figura 24: Terminal con mapa de memoria.
Fuente: s. a. *Dispositivos de entrada / salida* (s. f.: 51).

En las **pantallas mapeadas a caracteres***:*

- Cada carácter en la pantalla equivale a dos caracteres de RAM:
  - Uno aloja al código (ASCII) del caracter por exhibir.
  - Otro es el byte de atributo, necesario para determinar el color, el video inverso, el parpadeo, etc.

En las **terminales mapeadas a bits***:*

- Se utiliza el mismo principio.
- Cada bit en el video RAM controla en forma directa un solo pixel de la pantalla.
- Permite una completa flexibilidad en los tipos y tamaños de caracteres, varias ventanas y gráficos arbitrarios.

Con las pantallas mapeadas a memoria, el **teclado se desacopla totalmente de la pantalla***:*

- El teclado dispone de su propio manejador.
- El manejador del teclado puede operar en modo caracter o en modo línea.

Las terminales pueden operar con una **estructura central de buffers** o con **buffers exclusivos para cada terminal***.*

Frecuentemente **los manejadores de terminales soportan operaciones** tales como:

- Mover el cursor hacia arriba, abajo, a la izquierda o a la derecha una posición.
- Mover el cursor a x, y.
- Insertar un carácter o una línea en el cursor.
- Eliminar un carácter o una línea en el cursor.
- Recorrer la pantalla hacia arriba o hacia abajo "n" líneas.
- Limpiar la pantalla desde el cursor hacia el final de la línea o hasta el final de la pantalla.
- Trabajar en modo de video inverso, subrayado, parpadeo o normal.
- Crear, construir, mover o controlar las ventanas.

Software de entrada

El manejador del teclado:

El manejador del teclado pasa los caracteres del teclado a los programas de usuario. Su funcionamiento es el siguiente:

- **modo no interpretado (crudo):** Se limita a aceptar las entradas y pasarlas a un nivel superior sin modificación. El programa recibe todas las teclas: hpla←←← ola
- **modo interpretado (cocinado o canónico):** Está orientado a líneas. El manejador trata ciertos caracteres especiales, se encarga de la edición de cada línea y la entrega corregida a los programas de usuarios. El programa recibe la línea corregida: hola.

Software para la salida

- Salida en terminales RS-232
  - Hay buffers asociados a cada terminal.
  - La salida se copia primero en los buffers.
  - Tras copiar la salida, se envía el primer carácter y el manejador se duerme. Cuando llega una interrupción, se envía el siguiente carácter.
- Salidas en terminales con mapa en memoria
  - Los caracteres se colocan en la memoria de video.
  - Hay que tratar ciertos de forma especial (CR, LF, CTRL+G, etc.).
  - Se debe controlar la posición actual en la memoria de video. El siguiente caracter se coloca en esa posición y se avanza a la siguiente.

5.7. Interfaces gráficas de usuario

En el contexto del proceso de interacción persona-ordenador, la interfaz gráfica de usuario es el artefacto tecnológico de un sistema interactivo que posibilita, a través del uso y la representación del lenguaje visual, una interacción amigable con un sistema informático.

La interfaz gráfica de usuario (en Idioma inglés Graphical User Interface, GUI) es un tipo de interfaz de usuario que utiliza un conjunto de imágenes y objetos gráficos para representar la información y acciones disponibles en la interfaz. Habitualmente, las acciones se realizan mediante manipulación directa para facilitar la interacción del usuario con la computadora.

Surge como evolución de la línea de comandos de los primeros sistemas operativos y es pieza fundamental en un entorno gráfico. Como ejemplo de interfaz gráfica de usuario podemos citar el escritorio o *desktop* del sistema operativo Windows y el entorno X-Window de Linux y también Aqua de Mac OS X.

Hardware de teclado, ratón y pantalla para computadora personal.

Teclado: Es el dispositivo para introducción de datos por excelencia, el principal medio de comunicación entre el usuario y la computadora. Por medio de este periférico, los usuarios suministran órdenes, información, instrucciones, etc.

En las primeras computadoras personales, mucho antes del estándar PC, no se utilizaba el teclado tipo QWERTY que se utiliza en la actualidad. Este es similar a una máquina de escribir, aunque además de las teclas alfabéticas, numéricas y de puntuación, se incluyen símbolos y teclas de control. Además, su operación no es mecánica, sino que las teclas accionan interruptores que transmiten cierto código a la unidad central, donde se interpreta y ejecuta la acción respectiva.

El código al que está asociado cada carácter corresponde a un estándar conocido como código ASCII (se pronuncia «asqui»), por las siglas de *American Standard Code for Information Interchange*.

Ratón o mouse: Posiblemente usted ya ni conozca los sistemas operativos que trabajaban en modo texto, como el MS-DOS y el DR-DOS. La lógica de comunicación de este *software* operativo descansaba exclusivamente en instrucciones o comandos suministrados mediante el teclado. Pero, con el uso de menús desplegables en ciertos programas de aplicación de oficina y el posterior

advenimiento de los ambientes gráficos (Windows), el mouse o ratón se convirtió en un periférico indispensable en cualquier PC.

Este dispositivo permite apuntar, seleccionar y manipular áreas de trabajo en la pantalla para facilitar las operaciones informáticas del usuario, en vez de tener que escribir comandos de una sintaxis compleja. De hecho, hay actividades (como la ilustración y el tratamiento de imágenes) que simplemente no podrían efectuarse sin el ratón.

Las interfaces gráficas, con sus respectivos elementos lógicos (menús, iconos, ventanas, barras de desplazamiento, botones, cuadros de diálogo, etc.) que ahora nos son tan familiares, tuvieron sus orígenes en los laboratorios de Xerox, hacia principios de los años 1970, pero la tecnología no estaba a punto para poder aprovechar este medio de comunicación hombre-máquina tan sofisticado. Fue en los años 80, con la primera Macintosh, que las interfaces gráficas y su correspondiente dispositivo apuntador, comenzarían a ser una realidad cotidiana.

Monitor y pantallas planas: Así como el teclado es el puente básico para la comunicación del usuario con la computadora, ésta, a su vez, despliega sus resultados al usuario por medio del monitor.

El monitor lo único que hace es recibir la información que se envía desde la tarjeta madre y la convierte en puntos luminosos en la pantalla. Así que, estrictamente hablando, la resolución de una imagen y su profundidad de colores tienen que ver más con los circuitos de video y la memoria RAM adjudicada al video que con la calidad del monitor. Por esto, los fabricantes de computadoras insisten tanto en el tipo de tarjeta de video (circuitos, en realidad) incorporada en la tarjeta madre, y en la cantidad de memoria RAM asociada.

Actualmente existe una tendencia muy fuerte a sustituir los tradicionales monitores basados en tubos de rayos catódicos por pantallas planas tipo LCD (*display* de cristal líquido) o de plasma. Estas pantallas consumen menos energía, ocupan un menor espacio, y sobre todo no distorsionan la imagen.

Capítulo 6: Archivos

6.1. Archivos

Las reglas exactas para los nombres de archivos varían de sistema a sistema. Algunos sistemas de archivos distinguen entre las letras mayúsculas y minúsculas, mientras que otros no.

Muchos SO utilizan nombres de archivo con dos partes, separadas por un punto. La parte posterior al punto es la **extensión de archivo** y generalmente indica algo relativo al archivo, aunque las extensiones suelen ser meras convenciones.

Estructura de archivos:

Los archivos se pueden estructurar de varias maneras. Las más comunes son:

- **"secuencia de bytes":**
  - El archivo es una serie no estructurada de bytes.
  - Posee máxima flexibilidad.
  - El SO no ayuda pero tampoco estorba.
- **"secuencia de registros":**
  - El archivo es una secuencia de registros de longitud fija, cada uno con su propia estructura interna.
- **"árbol ":**
  - El archivo consta de un árbol de registros, no necesariamente de la misma longitud.
  - Cada registro tiene un campo *key* (llave o clave) en una posición fija del registro.
  - El árbol se ordena mediante el campo de clave para permitir una rápida búsqueda de una clave particular.

Tipos de archivos:

- Los **archivos regulares** son aquellos que contienen información del usuario.
- Los **directorios** son archivos de sistema para el mantenimiento de una estructura del sistema de archivos.

- Los **archivos especiales de caracteres**:
  - Tienen relación con la E/S.
  - Se utilizan para modelar dispositivos seriales de E/S (terminales, impresoras, redes, etc.).
- Los **archivos especiales de bloques** se utilizan para modelar discos.

Acceso a archivos:

Los tipos de acceso más conocidos son:

- **acceso secuencial:** El proceso lee en orden todos los registros del archivo comenzando por el principio, sin poder:
  - Saltar registros.
  - Leer en otro orden.

- **acceso aleatorio:** El proceso puede leer los registros en cualquier orden utilizando dos métodos para determinar el punto de inicio de la lectura:
  - Cada operación de lectura (*read*) da la posición en el archivo con la cual iniciar.
  - Una operación especial (seek) establece la posición de trabajo pudiendo luego leerse el archivo secuencialmente.

Atributos de archivos:

Cada archivo tiene:

- Su nombre y datos.
- Elementos adicionales llamados **atributos**, que varían considerablemente de sistema a sistema.

Algunos de los **posibles atributos** de archivo son:

- **"protección":** Quién debe tener acceso y de qué forma
- **"contraseña":** Contraseña necesaria para acceder al archivo
- **"creador":** Identificador de la persona que creó el archivo
- **"propietario":** Propietario actual
- **"bandera exclusivo - para - lectura":** 0 lectura / escritura, 1 para lectura exclusivamente
- **"bandera de ocultamiento":** 0 normal, 1 para no exhibirse en listas
- **"bandera de sistema":** 0 archivo normal, 1 archivo de sistema
- **"bandera de biblioteca":** 0 ya se ha respaldado, 1 necesita respaldo
- **"bandera ascii / binario":** 0 archivo en ascii, 1 archivo en binario

- **"bandera de acceso aleatorio":** 0 solo acceso secuencial, 1 acceso aleatorio
- **"bandera temporal":** 0 normal, 1 eliminar al salir del proceso
- **"Banderas de cerradura":** 0 no bloqueado, distinto de 0 bloqueado
- **"longitud del registro":** Número de bytes en un registro
- **"posición de la llave":** Ajuste de la llave dentro de cada registro
- **"longitud de la llave":** Número de bytes en el campo llave
- **"tiempo de creación":** Fecha y hora de creación del archivo
- **"tiempo del último acceso":** fecha y hora del último acceso al archivo
- **"tiempo de la última modificación":** Fecha y hora de la última modificación al archivo
- **"tamaño actual":** Número de bytes en el archivo
- **"tamaño máximo":** Tamaño máximo al que puede crecer el archivo

Operaciones con archivos:

Las llamadas más comunes al sistema relacionadas con los archivos son:

- ***create* (crear):** El archivo se crea sin datos.
- ***delete* (eliminar):** Si el archivo ya no es necesario debe eliminarse para liberar espacio en disco. Ciertos SO eliminan automáticamente un archivo no utilizado durante "n" días.
- ***open* (abrir):** Antes de utilizar un archivo, un proceso debe abrirlo. La finalidad es permitir que el sistema traslade los atributos y la lista de direcciones en disco a la memoria principal para un rápido acceso en llamadas posteriores.
- ***close* (cerrar):** Cuando concluyen los accesos, los atributos y direcciones del disco ya no son necesarios, por lo que el archivo debe cerrarse y liberar la tabla de espacio interno.
- ***read* (leer):** Los datos se leen del archivo. Quien hace la llamada debe especificar la cantidad de datos necesarios y proporcionar un buffer para colocarlos.
- ***write* (escribir):** Los datos se escriben en el archivo, en la posición actual. El tamaño del archivo puede aumentar (agregado de registros) o no (actualización de registros).
- ***append* (añadir):** Es una forma restringida de "*write*". Solo puede añadir datos al final del archivo.
- ***seek* (buscar):** Especifica el punto donde posicionarse. Cambia la posición del apuntador a la posición activa en cierto lugar del archivo.
- ***get attributes* (obtener atributos):** Permite a los procesos obtener los atributos del archivo.
- ***set attributes* (establecer atributos):** Algunos atributos pueden ser determinados por el usuario y modificados luego de la creación del

archivo. La información relativa al modo de protección y la mayoría de las banderas son un ejemplo obvio.
- ***rename* (cambiar de nombre):** Permite modificar el nombre de un archivo ya existente.

Archivos con correspondencia en memoria:

Algunos SO permiten asociar los archivos con un espacio de direcciones de un proceso en ejecución.

Se utilizan las llamadas al sistema **"*map*"** y **"*unmap*"**:

- **"*map*":** Utiliza un nombre de archivo y una dirección virtual. Hace que el SO asocie al archivo con la dirección virtual en el espacio de direcciones, por lo cual las lecturas o escrituras de las áreas de memoria asociadas al archivo se efectúan también sobre el archivo mapeado.
- **"*unmap*":** Elimina los archivos del espacio de direcciones y concluye la operación de asociación.

El mapeo de archivos elimina la necesidad de programar la E/S directamente, facilitando la programación.

Los principales problemas relacionados son:

- Imposibilidad de conocer a priori la longitud del archivo de salida, el que podría superar a la memoria.
- Dificultad para compartir los archivos mapeados evitando inconsistencias, ya que las modificaciones hechas en las páginas no se verán reflejadas en el disco hasta que dichas páginas sean eliminadas de la memoria.

6.2. Directorios

Generalmente son utilizados por los SO para llevar un registro de los archivos. En muchos sistemas son a su vez también archivos.

Sistemas de directorios de un solo nivel

Es la estructura más sencilla de directorios. Todos los archivos están ubicados en un solo directorio. El sistema operativo CP/M es un ejemplo de este caso. La principal deficiencia es cuando el número de archivos crecía o cuando más de un usuario utilizaba el sistema.

Sistemas de directorios de dos niveles

En este esquema, cada usuario tiene su propio directorio, contrario al esquema de un solo nivel. Existía un directorio de usuario donde yacían sus propios archivos. El directorio de nivel superior no es más que una tabla indexada por el nombre o número de identificación de usuario, denominado DMF (**directorio maestro de ficheros**).

Sistemas de directorios jerárquicos

El directorio contiene un conjunto de datos por cada archivo referenciado.

Una posibilidad es que el directorio contenga por cada archivo referenciado:

- El nombre
- Sus atributos
- Las direcciones en disco donde se almacenan los datos

Otra posibilidad es que cada entrada del directorio contenga:

- El nombre del archivo
- Un apuntador a otra estructura de datos donde se encuentran los atributos y las direcciones en disco

Al abrir un archivo el *SO*:

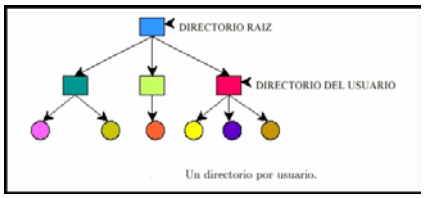
- Busca en su directorio el nombre del archivo
- Extrae los atributos y direcciones en disco
- Graba esta información en una tabla de memoria real
- Todas las referencias subsecuentes al archivo utilizarán la información de la memoria principal

El número y organización de directorios varía de sistema en sistema:

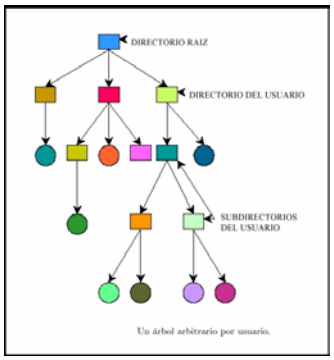
- **directorio único:** El sistema tiene un solo directorio con todos los archivos de todos los usuarios.
- **un directorio por usuario:** El sistema habilita un solo directorio por cada usuario
- **un árbol de directorios por usuario:** El sistema permite que cada usuario tenga tantos directorios como necesite, respetando una jerarquía general.

A continuación, se muestran estos tres últimos esquemas:

a) Directorio único



b) Directorio con usuario



c) Árbol de directorio por usuario

Figura 25: Esquema de directorios.
Fuente: Tanenbaum (2003: 393, 394,395).

Nombres de rutas:

Cuando el sistema de archivos está organizado como un árbol de directorios se necesita una forma de determinar los nombres de los archivos.

Los principales métodos para nombres de los archivos son:

- **ruta de acceso absoluta:**
  - Cada archivo tiene una ruta de acceso absoluta.
  - Consta de la ruta de acceso desde el directorio raíz hasta el archivo.
  - Los componentes de la ruta de acceso se separan mediante algún carácter llamado *"***separador***"*.

- **ruta de acceso relativa:**
  - Se utiliza junto con el concepto de directorio de trabajo o directorio activo.
  - Todos los nombres que no comiencen en el directorio raíz se toman en relación con el directorio de trabajo.
  - El nombre absoluto de la ruta de acceso siempre funciona, sin importar cuál sea el directorio de trabajo.

Operaciones con directorios:

Las llamadas al sistema permitidas para el manejo de los directorios tienen variación de sistema a sistema:
Las más comunes son las siguientes:

- ***create* (crear):** Se crea un directorio vacío.
- ***delete* (eliminar):** Se elimina un directorio, que debe estar vacío.
- ***opendir* (abrir directorio):** se pueden leer los directorios:
  - Antes de poder leer un directorio, éste debe ser abierto.
- ***closedir* (cerrar directorio):** Cuando se ha leído un directorio, éste debe ser cerrado para liberar el espacio correspondiente de la tabla interna.
- ***readdir* (leer directorio):** Regresa la siguiente entrada en un directorio abierto, sin importar el tipo de estructura de directorios que se utilice.
- **rename (cambiar de nombre):** Cambia el nombre de un directorio de manera similar al cambio para archivos.
- ***link* (ligar):** Es una técnica que permite que un archivo aparezca en más de un directorio:
  - Especifica un archivo existente y el nombre de una ruta de acceso.
  - Crea un enlace del archivo ya existente con el nombre especificado en la ruta de acceso.
- ***unlink* (desligar):** Se elimina una entrada del directorio:
  - Si el archivo que se desea desligar aparece solo en un directorio (el caso normal):
    - Se elimina del sistema de archivos.
  - Si el archivo que se desea desligar está presente en varios directorios:
    - Solo se elimina la ruta de acceso especificada.
    - Las demás rutas permanecen.

6.3. Implementación de sistemas de archivos

Organización de sistemas de archivos:

Se consideran aspectos tales como:

- La forma de almacenamiento de archivos y directorios
- La administración del espacio en disco
- La forma de hacerlo de manera eficiente y confiable

Se deben tener presentes problemas tales como la "fragmentación" creciente del espacio en disco:

- Ocasiona problemas de performance al hacer que los archivos se desperdiguen a través de bloques muy dispersos.
- Una técnica para aliviar el problema de la "fragmentación" consiste en realizar periódicamente:
    - **"Condensación":** Se pueden "reorganizar" los archivos expresamente o automáticamente según algún criterio predefinido.
    - **"Recolección de basura o residuos":** Se puede hacer fuera de línea o en línea, con el sistema activo, según la implementación.

Implementación de archivos:

El aspecto clave de la implantación del almacenamiento de archivos es el registro de los bloques asociados a cada archivo.

Algunos de los métodos utilizados son los siguientes:

- *Asignación contigua o adyacente:*

    - Los archivos son asignados a áreas contiguas de almacenamiento secundario.
    - Las principales **ventajas** son:
        - Facilidad de implantación, ya que solo se precisa el número del bloque de inicio para localizar un archivo.
        - Rendimiento excelente respecto de la E/S.
    - Los principales **defectos** son:
        - Se debe conocer el tamaño máximo del archivo al crearlo.
        - Produce una gran fragmentación de los discos

- *Asignación no contigua:*

    - Son esquemas de almacenamiento más dinámicos, destacándose los siguientes:
    - *Asignación encadenada orientada hacia el sector:*
        - El disco se considera compuesto de sectores individuales.
        - Los archivos constan de varios sectores que pueden estar dispersos por todo el disco.
            - Los sectores que pertenecen a un archivo común contienen apuntadores de uno a otro formando una *"lista encadenada".*

        - Una *"lista de espacio libre"* contiene entradas para todos los sectores libres del disco.
        - Las ampliaciones o reducciones en el tamaño de los archivos se resuelven actualizando la *"lista de espacio libre"* y no hay necesidad de condensación.
        - Las principales **desventajas** son:
            - Debido a la posible dispersión en el disco, la recuperación de registros lógicamente contiguos puede significar largas búsquedas.
            - El mantenimiento de la estructura de *"listas encadenadas"* significa una sobrecarga en tiempo de ejecución.
            - Los apuntadores de la estructura de lista consumen espacio en disco.
- *Asignación por bloques:*
    - Es más eficiente y reduce la sobrecarga en ejecución.
    - Es una mezcla de los métodos de asignación contigua y no contigua.
    - Se asignan bloques de sectores contiguos en vez de sectores individuales.
    - El sistema trata de asignar nuevos bloques a un archivo eligiendo bloques libres lo más próximos posible a los bloques del archivo existentes.
    - Las formas más comunes de implementar la asignación por bloques son:
        - Encadenamiento de bloques.
        - Encadenamiento de bloques de índice.
        - Transformación de archivos orientada hacia bloques.
- *Encadenamiento de bloques o lista ligada:*
    - Las entradas en el directorio de usuarios apuntan al primer bloque de cada archivo.
    - Cada uno de los bloques de longitud fija que forman un archivo contiene dos partes:
        - Un bloque de datos.
        - Un apuntador al bloque siguiente.
    - Cada bloque contiene varios sectores.
    - Frecuentemente el tamaño de un bloque se corresponde con el de una pista completa del disco.
    - Localizar un registro determinado requiere:
        - Buscar en la cadena de bloques hasta encontrar el bloque apropiado.
        - Buscar en el bloque hasta encontrar el registro.

        - El examen de la cadena desde el principio puede ser lento ya que debe realizarse de bloque en bloque, y pueden estar dispersos por todo el disco.
        - La inserción y el retiro son inmediatos, dado que se deben modificar los apuntadores del bloque precedente.
        - Se pueden usar *"listas de encadenamiento doble"*, hacia adelante y hacia atrás, con lo que se facilita la búsqueda.
    - *Encadenamiento de bloques de índices:*
        - Los apuntadores son colocados en varios bloques de índices separados:
            - Cada bloque de índices contiene un número fijo de elementos.
            - Cada entrada contiene un *identificador de registros* y un *apuntador* a ese registro.
            - Si es necesario utilizar más de un bloque de índices para describir un archivo, se encadena una serie de bloques de índices.
        - La gran **ventaja** es que la búsqueda puede realizarse en los propios bloques de índices.
        - Los bloques de índices pueden mantenerse juntos en el almacenamiento secundario para acortar la búsqueda, pero para mejor performance podrían mantenerse en el almacenamiento primario.
        - La principal **desventaja** es que las inserciones pueden requerir la reconstrucción completa de los bloques de índices:
            - Una posibilidad es dejar vacía una parte de los bloques de índices para facilitar inserciones futuras y retardar las reconstrucciones.
        - Es suficiente que el dato del directorio contenga el número de bloque inicial para localizar todos los bloques restantes, sin importar el tamaño del archivo.

Implementación de directorios:

Para abrir un archivo el SO utiliza información del directorio:

- El directorio contiene la información necesaria para encontrar los bloques en el disco.
- El tipo de información varía según el sistema.

La principal función del sistema de directorios es asociar el nombre del archivo con la información necesaria para localizar los datos.

Un aspecto íntimamente ligado con esto es la posición de almacenamiento de los **atributos:**

- Una posibilidad es almacenarlos en forma directa **dentro del dato del directorio**.
- Otra posibilidad es almacenar los atributos en el *nodo-i* en vez de utilizar la entrada del directorio.

Archivos compartidos:

Frecuentemente conviene que los archivos compartidos aparezcan simultáneamente en distintos directorios de diferentes usuarios.

El propio sistema de archivos es una **gráfica dirigida acíclica** en vez de un árbol.

La conexión entre un directorio y un archivo de otro directorio, el cual comparten, se denomina **enlace**.

Si los directorios realmente contienen direcciones en disco:

- Se debe tener una copia de las direcciones en disco en el directorio que accede al archivo compartido al enlazar el archivo.
- Se debe evitar que los cambios hechos por un usuario a través de un directorio no sean visibles por los demás usuarios, para lo que se consideraran dos soluciones posibles.

Primera solución:

- Los bloques del disco no se enlistan en los directorios, sino en una pequeña estructura de datos asociada al propio archivo.
- Los directorios apuntarían solo a esa pequeña estructura de datos, que podría ser el nodo-i.

Segunda solución:

- El enlace se produce haciendo que el sistema cree un nuevo archivo de tipo *"link".*
- El archivo *"link":*
    - Ingresa al directorio del usuario que accede a un archivo de otro directorio y usuario.
    - Solo contiene el *nombre de la ruta de acceso* del archivo al cual se enlaza.
- Este criterio se denomina enlace simbólico.

Desventajas de la primera solución:

- La creación de un enlace:
  - No modifica la propiedad respecto de un archivo.
  - Aumenta el contador de enlaces del nodo-i:
    - El sistema sabe el número de entradas de directorio que apuntan en cierto momento al archivo.
- Si el propietario inicial del archivo intenta eliminarlo, surge un problema para el sistema:
  - Si elimina el archivo y limpia el nodo-i, el directorio que enlazó al archivo tendrá una entrada que apunta a un nodo-i no válido.
  - Si el nodo-i se reasigna a otro archivo el enlace apuntará al archivo incorrecto.
  - El sistema:
    - Puede ver por medio del contador de enlaces en el nodo-i que el archivo sigue utilizándose.
    - No puede localizar todas las entradas de directorio asociadas a ese archivo para eliminarlas.

- La solución podría ser:
  - Eliminar la entrada del directorio inicialmente propietario del archivo.
  - Dejar intacto el nodo-i:
    - Se daría el caso que el directorio que posee el enlace es el único que posee una entrada de directorio para un archivo de otro directorio, para el cual dicho archivo ya no existe.
    - Esto no ocurre con los enlaces simbólicos ya que solo el propietario verdadero tiene un apuntador al nodo-i:
      - Los usuarios enlazados al archivo solo tienen nombres de rutas de acceso y no apuntadores a nodo-i.
      - Cuando el propietario elimina un archivo, este se destruye.

Desventajas de la segunda solución:

- El principal problema es su costo excesivo, especialmente en accesos a disco, puesto que se debe leer el archivo que contiene la ruta de acceso, analizarla y seguirla componente a componente hasta alcanzar el nodo-i.
- Se precisa un nodo-i adicional por cada enlace simbólico y un bloque adicional en disco para almacenar la ruta de acceso.

- Los archivos pueden tener dos o más rutas de acceso, debido a lo cual en búsquedas genéricas se podría encontrar el mismo archivo por distintas rutas y tratárselo como si fueran archivos distintos.

Los enlaces simbólicos tienen la ventaja de que se pueden utilizar para enlazar archivos en otras máquinas, en cualquier parte del mundo; se debe proporcionar solo la dirección de la red de la máquina donde reside el archivo y su ruta de acceso en esa máquina.

Administración del espacio en disco:

Existen dos *estrategias generales* para almacenar un archivo de *"n"* bytes:

Asignar *"n"* bytes consecutivos de espacio en el disco:

- Tiene el problema de que si un archivo crece será muy probable que deba desplazarse en el disco, lo que puede afectar seriamente al rendimiento.

Dividir el archivo en cierto número de bloques (no necesariamente) adyacentes:

- Generalmente los sistemas de archivos utilizan esta estrategia con bloques de tamaño fijo.

Tamaño del bloque:

Dada la forma en que están organizados los bloques, el sector, la pista y el cilindro son los candidatos obvios como unidades de asignación.

Si se tiene una **unidad de asignación grande**, como un cilindro, esto significa que cada archivo, inclusive uno pequeño, ocupará todo un cilindro; con esto se desperdicia espacio de almacenamiento en disco.

Si se utiliza una **unidad de asignación pequeña**, como un sector, implica que cada archivo constará de muchos bloques; con esto su lectura generará muchas operaciones de E/S afectando la performance.

Lo anterior indica que **la eficiencia en tiempo y espacio tienen un conflicto inherente***.*

Generalmente se utilizan como solución de compromiso bloques de 1/2 k, 1k, 2k o 4k (Ver la siguiente imagen).

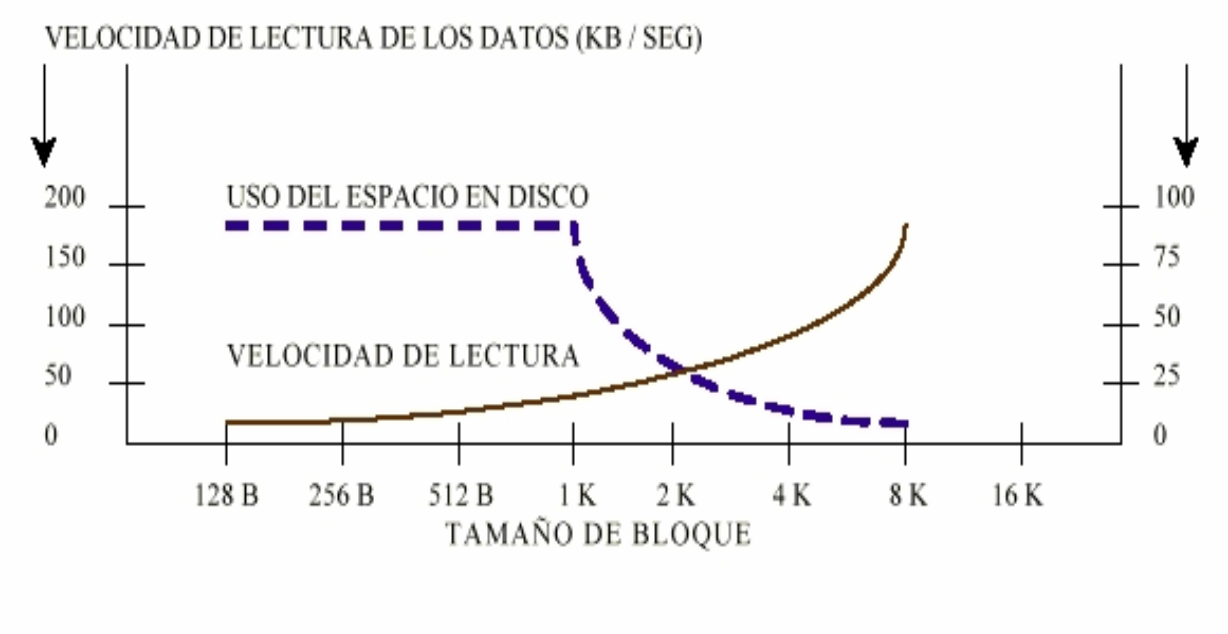


Figura 26: Representación de la velocidad de lectura y del uso del espacio en disco en función del tamaño de bloque.
Fuente: Tanenbaum (2003: 412).

Hay que recordar que el **tiempo de lectura de un bloque de disco** es la suma de los tiempos de:

- Búsqueda
- Demora rotacional
- Transferencia

Registro de los bloques libres:

Se utilizan por lo general dos métodos:

- La lista de bloques libres como lista ligada
- Un mapa de bits

Lista ligada de bloques de disco:

- Cada bloque contiene tantos números de bloques libres como pueda.
- Los bloques libres se utilizan para contener a la lista de bloques libres.

Mapa de bits:

- Un disco con *"n"* bloques necesita un mapa de bits con *"n"* bits.
- Los bloques libres se representan con "1" y los asignados con "0" (o viceversa).

- Generalmente este método es preferible cuando existe espacio suficiente en la memoria principal para contener *completo el mapa de bits.*

Disk quotas:

Para evitar que los usuarios se apropien de un espacio excesivo en disco, los SO multiusuario proporcionan generalmente un mecanismo para establecer las cuotas en el disco.

La idea es que:

- Un administrador del sistema asigne a cada usuario una proporción máxima de archivos y bloques.
- El SO garantice que los usuarios no excedan sus cuotas.

Un mecanismo utilizado es el siguiente:

- Cuando un usuario **abre un archivo:**
  - Se localizan los atributos y direcciones en disco.
  - Se colocan en una tabla de archivos abiertos en la memoria principal.
  - Uno de los atributos indica el propietario del archivo; cualquier aumento del tamaño del archivo se carga a la cuota del propietario.
  - Una segunda tabla contiene el registro de las cuotas para cada uno de los usuarios que tengan un archivo abierto en ese momento, aún cuando el archivo lo haya abierto otro usuario.
- Cuando se escribe una **nueva entrada en la tabla de archivos abiertos:**
  - Se introduce un apuntador al registro de la cuota del propietario para localizar los límites.
- Cuando se **añade un bloque a un archivo:**
  - Se incrementa el total de bloques cargados al propietario.
  - Se verifica este valor contra los límites estricto y flexible (el primero no se puede superar, el segundo sí).
  - También se verifica el número de archivos.

Confiabilidad del sistema de archivos

Es necesario proteger la información alojada en el sistema de archivos, efectuando los resguardos correspondientes. De esta manera se evitan las

consecuencias generalmente catastróficas de la pérdida de los sistemas de archivos.

Las pérdidas se pueden deber a problemas de *hardware*, *software*, hechos externos, etc.

Manejo de un bloque defectuoso:

Se utilizan soluciones por *hardware* y por *software*.

La **solución en *hardware*:**

- Consiste en dedicar un sector del disco a la *lista de bloques defectuosos.*
- Al inicializar el controlador por primera vez:
  - Lee la *"lista de bloques defectuosos".*
  - Elige un bloque (o pista) de reserva para reemplazar los defectuosos.
  - Registra la asociación en la lista de bloques defectuosos.
  - En lo sucesivo, las solicitudes del bloque defectuoso utilizarán el de repuesto.

La **solución en *software*:**

- Requiere que el usuario o el sistema de archivos construyan un *archivo con todos los bloques defectuosos.*
- Se los elimina de la *"lista de bloques libres".*
- Se crea un *"archivo de bloques defectuosos":*
  - Está constituido por los bloques defectuosos.
  - No debe ser leído ni escrito.
  - No se debe intentar obtener copias de respaldo de este archivo.

Respaldos (copias de seguridad o de back-up):

Es muy importante respaldar los archivos con frecuencia. Los respaldos pueden consistir en efectuar copias completas del contenido de los discos (flexibles o rígidos). Una estrategia de respaldo consiste en dividir los discos en **áreas de datos** y **áreas de respaldo**, utilizándolas de a pares:

- Se desperdicia la mitad del almacenamiento de datos en disco para respaldo.
- Cada noche (o en el momento que se establezca), la parte de datos de la unidad 0 se copia a la parte de respaldo de la unidad 1 y viceversa.

Otra estrategia es el vaciado por incrementos o **respaldo incremental***:*

- Se obtiene una copia de respaldo periódicamente (por ejemplo, una vez por mes o por semana), llamada copia total.
- Se obtiene una copia diaria solo de aquellos archivos modificados desde la última copia total. En estrategias mejoradas, se copian solo aquellos archivos modificados desde la última vez que dichos archivos fueron copiados.
- Se debe mantener en el disco información de control como una *"lista de los tiempos de copiado"* de cada archivo, la que debe ser actualizada cada vez que se obtienen copias de los archivos y cada vez que los archivos son modificados.
- Puede requerir una gran cantidad de cintas de respaldo dedicadas a los respaldos diarios entre respaldos completos.

Consistencia del sistema de archivos*:*

Muchos sistemas de archivos leen bloques, los modifican y escriben en ellos después.

Si el sistema falla antes de escribir en los bloques modificados, el sistema de archivos puede quedar en un *"estado inconsistente".*

La inconsistencia es particularmente crítica si alguno de los bloques afectados es:

- Bloques de nodos-i
- Bloques de directorios
- Bloques de la lista de bloques libres

La mayoría de los sistemas dispone de un programa utilitario que verifica la **consistencia del sistema de archivos***:*

- Se pueden ejecutar al arrancar el sistema o a pedido.
- Pueden actuar sobre todos o algunos de los discos.
- Pueden efectuar verificaciones a nivel de bloques y a nivel de archivos.
- La consistencia del sistema de archivos no asegura la consistencia interna de cada archivo, respecto de su contenido.
- Generalmente pueden verificar también el sistema de directorios y / o de bibliotecas.

Generalmente los utilitarios utilizan dos tablas:

- Tabla de bloques en uso
- Tabla de bloques libres
- Cada bloque debe estar referenciado en una de ellas

Si un bloque no aparece en ninguna de las tablas se trata de una falla llamada **bloque faltante***:*

- No produce daños pero desperdicia espacio en disco.
- Se soluciona añadiendo el bloque a la tabla de bloques libres.

También podría detectarse la situación de falla debida a un **bloque referenciado dos veces en la tabla de bloques libres**:

- Esta falla no se produce en los sistemas de archivos basados en mapas de bits, sí en los basados en tablas o listas.
- La solución consiste en depurar la tabla de bloques libres.

Una **falla muy grave** es que **el mismo bloque de datos aparezca referenciado dos o más veces en la tabla de bloques en uso**:

- Como parte del mismo o de distintos archivos.
- Si uno de los archivos se borra, el bloque aparecería en la tabla de bloques libres y también en la de bloques en uso.
- Una solución es que el verificador del sistema de archivos:
    - Asigne un bloque libre.
    - Copie en el bloque libre el contenido del bloque conflictivo.
    - Actualice las tablas afectando el bloque copia a alguno de los archivos.
    - Agregue el bloque conflictivo a la tabla de bloques libres.
    - Informe al usuario para que verifique el daño detectado y la solución dada.

Otro error posible es que **un bloque esté en la tabla de bloques en uso y en la tabla de bloques libres**:

- Se soluciona eliminándolo de la tabla de bloques libres.

Las verificaciones de directorios incluyen controles como:

- Número de directorios que apuntan a un nodo-i con los contadores de enlaces almacenados en los propios nodos-i; en un sistema consistente de archivos deben coincidir.

Una posible falla es que **el contador de enlaces sea mayor que el número de entradas del directorio**:

- Aunque se eliminaran todos los archivos de los directorios el contador sería distinto de cero y no se podría eliminar el nodo-i.
- No se trata de un error serio pero produce desperdicio de espacio en disco con archivos que no se encuentran en ningún directorio.
- Se soluciona haciendo que el contador de enlaces en el nodo-i tome el valor correcto; si el valor correcto es 0, el archivo debe eliminarse.

Otro tipo de **error** es potencialmente **catastrófico**:

- Si dos entradas de un directorio se enlazan a un archivo, pero el nodo-i indica que solo existe un enlace, entonces, al eliminar cualquiera de estas entradas de directorio, el contador del nodo-i tomará el valor 0.
- Debido al valor 0 el sistema de archivos lo señala como no utilizado y libera todos sus bloques.
- Uno de los directorios apunta hacia un nodo-i no utilizado, cuyos bloques se podrían asignar entonces a otros archivos.
- La solución es forzar que el contador de enlaces del nodo-i sea igual al número de entradas del directorio.

También se pueden hacer **verificaciones heurísticas**, por ejemplo:

- Cada nodo-i tiene un modo, pero algunos modos son válidos aunque extraños:
  - Ejemplo: Se prohíbe el acceso al propietario y todo su grupo, pero se permite a los extraños leer, escribir y ejecutar el archivo.
  - La verificación debería detectar e informar de estas situaciones.
- Se debería informar como sospechosos aquellos directorios con excesivas entradas, por ejemplo, más de mil.

Desempeño del sistema de archivos

El acceso al disco es mucho más lento que el acceso a la memoria:

- Los tiempos se miden en milisegundos y en nanosegundos respectivamente.
- Se debe reducir el número de accesos a disco.

La técnica más común para **reducir** *los accesos a disco* es el **bloque caché** o **buffer caché.**

- Se utiliza el término **ocultamiento** para esta técnica (del francés "*cacher*": ocultar).
- Un **caché** es una colección de bloques que pertenecen desde el punto de vista lógico al disco, pero que se mantienen en memoria por razones de rendimiento.

Uno de los algoritmos más comunes para la administración del caché es el siguiente:

- Verificar todas las solicitudes de lectura para saber si el bloque solicitado se encuentra en el caché.
- En caso afirmativo, se satisface la solicitud sin un acceso a disco.
- En caso negativo, se lee para que ingrese al caché y luego se copia al lugar donde se necesite.
- Cuando hay que cargar un bloque en un caché totalmente ocupado:
    - Hay que eliminar algún bloque y volverlo a escribir en el disco en caso de que haya sido modificado luego de haberlo traído del disco.
    - Se plantea una situación muy parecida a la paginación y se resuelve con algoritmos similares.

Se debe considerar la **posibilidad de una falla total del sistema y su impacto en la consistencia del sistema de archivos***:*

- Si un bloque crítico, como un bloque de un nodo-i, se lee en el caché y se modifica, sin volverse a escribir en el disco, una falla total del sistema dejará al sistema de archivos en un estado inconsistente.

Se deben tener en cuenta los siguientes factores:

- ¿Es posible que el bloque modificado se vuelva a necesitar muy pronto?:
    - Los bloques que se vayan a utilizar muy pronto, como un bloque parcialmente ocupado que se está escribiendo, deberían permanecer un *"largo tiempo"*.
- ¿Es esencial el bloque para la consistencia del sistema de archivos?:
    - Si es esencial (generalmente lo será si no es bloque de datos) y ha sido modificado, debe escribirse en el disco de inmediato:
        - Se reduce la probabilidad de que una falla total del sistema haga naufragar al sistema de archivos.
        - Se debe elegir con cuidado el orden de escritura de los bloques críticos.
- No es recomendable mantener los bloques de datos en el caché durante mucho tiempo antes de reescribirlos.

La solución de algunos SO consiste en tener **una llamada al sistema que fuerza una actualización general** a intervalos regulares de algunos segundos (por ejemplo 30).

Otra solución consiste en escribir los bloques modificados (del caché) al disco, tan pronto como haya sido escrito (el caché):

- Se dice que se trata de **cachés de escritura.**
- Requiere más E/S que otros tipos de cachés.

Una técnica importante para **aumentar el rendimiento de un sistema de archivos** es la reducción de la cantidad de movimientos del brazo del disco (mecanismo de acceso):

- Se deben colocar los bloques que probablemente tengan un acceso secuencial, próximos entre sí, preferentemente en el mismo cilindro.
- Los nodos-i deben estar a mitad del disco y no al principio, reduciendo a la mitad el tiempo promedio de búsqueda entre el nodo-i y el primer bloque del archivo.

Capítulo 9: Seguridad

9.1. El entorno de la seguridad

Los sistemas de archivos generalmente contienen información muy valiosa para sus usuarios, razón por la que los sistemas de archivos deben protegerla. Asimismo, el *hardware* que sirve de soporte a la ejecución de tareas, también debe ser cubierto por la seguridad del sistema operativo.

Amenazas

Los sistemas operativos deben garantizar seguridad ante múltiples amenazas que se producen en el computador. Entre estas amenazas se incluyen todas aquellas que atentan contra la integridad y la funcionalidad de archivos y partes electrónicas del equipo. Todas estas amenazas pueden ser de origen interno o externo.

Intrusos

Respecto del problema de los intrusos, se los puede clasificar como:

- Pasivos: Solo desean leer archivos que no están autorizados a leer.
- Activos: Desean hacer cambios no autorizados a los datos.

Para diseñar un sistema seguro contra intrusos:

- Hay que tener en cuenta el tipo de intrusos contra los que se desea tener protección.
- Hay que ser consciente de que la cantidad de esfuerzo que se pone en la seguridad y la protección depende claramente de quién se piensa sea el enemigo.

Algunos **tipos de intrusos** son los siguientes:

- Curiosidad casual de usuarios no técnicos
- Conocidos (técnicamente capacitados) husmeando
- Intentos deliberados por hacer dinero

Pérdida accidental de datos

La pérdida accidental de datos es una situación en la cual el sistema operativo debe pensar y dotar al usuario de medidas contingentes para que se pueda recuperar.

Ante estas situaciones el sistema operativo deberá permitir manejar herramientas que permitan el respaldo de información, seguridad en el acceso de archivos, perfiles de administración del equipo, etc.

9.2. Aspectos básicos de criptografía

La **criptografía** permite convertir un archivo o mensaje de texto simple y convertirlo en texto cifrado, de tal forma que solo las personas autorizadas puedan restaurarlo a su fuente original. Criptografía es el estudio de las técnicas matemáticas relativas a los aspectos de seguridad e información; tales técnicas abarcan: confidencialidad, integridad de los datos, autenticación de la entidad, y autenticación del origen de los datos. Sin embargo, la criptografía no pretende proveer los medios para asegurar la información, sino ofrecer las técnicas para lograr este aseguramiento.

El cifrado simétrico (también conocido como **cifrado de clave privada** o **cifrado de clave secreta**) consiste en utilizar la misma clave para el cifrado y el descifrado.

Funciones unidireccionales son funciones que tienen la propiedad de ser fáciles de calcular pero difíciles de invertir. **Fácil de calcular** se refiere a que un algoritmo la puede computar en tiempo polinomial, en función del largo de la entrada. **Difícil de invertir** significa que no hay algoritmo probabilístico que en tiempo polinomial puede computar una preimagen de f(x) cuando x es escogido al azar. Algunas personas conjeturan que el logaritmo discreto y la inversión RSA son funciones de un solo sentido.

Firmas digitales es, en la transmisión de mensajes telemáticos y en la gestión de documentos electrónicos, un método criptográfico que asocia la **identidad** de una persona o de un equipo informático al mensaje o documento. En función del tipo de firma, puede, además, asegurar la **integridad** del documento o mensaje.

La **firma electrónica**, como la firma **ológrafa** (autógrafa, manuscrita), puede vincularse a un documento para identificar al autor, para señalar conformidad (o disconformidad) con el contenido, para indicar que se ha

leído o, según el tipo de firma, garantizar que no se pueda modificar su contenido.

9.3. Autenticación de usuarios

Autenticación de contraseñas

Las clases de elementos de autentificación para establecer la identidad de una persona son:

- Algo sobre la persona: como huellas digitales, registro de la voz, fotografía, firma, etc.
- Algo poseído por la persona: como insignias especiales, tarjetas de identificación, llaves, etc.
- Algo conocido por la persona: como contraseñas, combinaciones de cerraduras, etc.

El esquema más común de autentificación es la **protección por contraseña**:

- El usuario elige una palabra clave, la memoriza, la teclea para ser admitido en el sistema computarizado:
  - La clave no debe desplegarse en pantalla ni aparecer impresa.

La protección por contraseñas tiene ciertas desventajas si no se utilizan criterios adecuados para:

- Elegir las contraseñas
- Comunicarlas fehacientemente en caso de que sea necesario
- Destruir las contraseñas luego de que han sido comunicadas
- Modificarlas luego de algún tiempo

Los usuarios tienden a elegir contraseñas fáciles de recordar:

- Nombre de un amigo, pariente, perro, gato, etc.
- Número de documento, domicilio, patente del auto, etc.

Estos datos podrían ser conocidos por quien intente una violación a la seguridad mediante intentos repetidos. Por lo tanto, debe limitarse la cantidad de intentos fallidos de acierto para el ingreso de la contraseña.

La contraseña no debe ser muy corta para no facilitar la probabilidad de acierto. Tampoco debe ser muy larga para que no se dificulte su

memorización, ya que los usuarios la anotarían por miedo a no recordarla, y ello incrementaría los riesgos de que trascienda.

Autenticación biométrica

Es la técnica que estudia las características físicas de una persona, tales como huellas dactilares, geometría de la mano, estructura del ojo o patrón de la voz.

9.4. Ataques desde dentro del sistema

Caballos de Troya

También debe señalarse la posibilidad del ataque del caballo de Troya:

- Modificar un programa normal para que haga cosas adversas además de su función usual.
- Arreglar las cosas para que la víctima utilice la versión modificada.

Falsificación de inicios de sesión

Esta técnica consiste en tomar la identidad de otro usuario y actuar en su nombre. Una forma común es acceder al equipo como un usuario legítimo, suplantando la interface de inicio de sesión al sistema, y apoderarse de la información contenida en su equipo.

Bombas lógicas

Es un programa informático que se instala en un ordenador y permanece oculto hasta cumplirse una o más condiciones preprogramadas, para entonces ejecutar una acción.

A diferencia de un virus, una bomba lógica jamás se reproduce por sí sola.

Ejemplos de condiciones predeterminadas:

- Día de la semana concreto
- Hora concreta

Trampas

La intención de las trampas informáticas es atraer a *crackers* o *spammers*, simulando ser sistemas vulnerables o débiles a los ataques. Es una

herramienta de seguridad informática utilizada para recoger información sobre los atacantes y sus técnicas. Los *honeypots* pueden distraer a los atacantes de las máquinas más importantes del sistema, y advertir rápidamente al administrador del sistema de un ataque, además de permitir un examen en profundidad del atacante, durante y después del ataque al *honeypot*.

9.5. Ataques desde afuera del sistema

Virus

Es un **malware** que tiene por objeto alterar el normal funcionamiento de la computadora, sin el permiso o el conocimiento del usuario. Habitualmente, los virus reemplazan archivos ejecutables por otros infectados con el código de este. Los virus pueden destruir, de manera intencionada, los datos almacenados en un ordenador, aunque también existen otros más "benignos", que solo se caracterizan por ser molestos.

Los virus informáticos tienen básicamente la función de propagarse. No se replican a sí mismos porque no tienen esa facultad, como el gusano informático. Dependen de un *software* para propagarse, son muy dañinos y algunos contienen además una carga dañina (*payload*) con distintos objetivos: desde una simple broma hasta realizar daños importantes en los sistemas, o bloquear las redes informáticas generando tráfico inútil.

Dado que una característica de los virus es el consumo de recursos, los virus ocasionan problemas tales como pérdida de productividad, cortes en los sistemas de información o daños a nivel de datos.

Otra de las características es la posibilidad que tienen de ir **replicándose**. Las redes en la actualidad ayudan a dicha propagación cuando éstas no tienen la seguridad adecuada.

Otros daños que los virus producen a los sistemas informáticos son la pérdida de información, horas de parada productiva, tiempo de reinstalación, etc.

Hay que tener en cuenta que cada virus plantea una situación diferente.

Los métodos para disminuir o reducir los riesgos asociados a los virus pueden ser los denominados activos o pasivos.

Activos

- Usar sistemas operativos más seguros que Windows como GNU/Linux, Mac OS o Freebsd.

- **Antivirus**: Los llamados programas antivirus tratan de descubrir las trazas que ha dejado un *software* malicioso, para detectarlo y eliminarlo, y en algunos casos contener o parar la contaminación. Tratan de tener controlado el sistema mientras funciona, parando las vías conocidas de infección y notificando al usuario de posibles incidencias de seguridad.

- **Filtros de ficheros**: Consiste en generar filtros de ficheros dañinos si el ordenador está conectado a una red. Estos filtros pueden usarse, por ejemplo, en el sistema de correos o usando técnicas de firewall. En general, este sistema proporciona una seguridad donde no se requiere la intervención del usuario. Puede ser muy eficaz, y permitir emplear únicamente recursos de forma más selectiva.

Worms (gusanos)

Se registran para correr cuando inicia el sistema operativo ocupando la memoria y volviendo lento al ordenador, pero no se adhieren a otros archivos ejecutables. Utilizan medios masivos como el correo electrónico para esparcirse de manera global.

Código móvil:

Conforme crece la demanda de teléfonos inteligentes más sofisticados, se espera que los cibercriminales apunten al ambiente móvil. Aunque los dispositivos basados en Linux y Windows son cada vez más populares, el SO Symbian sigue siendo el sistema operativo preferido por los usuarios móviles y fabricantes en todo el mundo – y probablemente siga siendo el principal objetivo de los autores de amenazas. Hasta ahora, el código malicioso basado en Symbian encabeza las listas de infecciones, con las viejas familias del código malicioso SYMBOS como Comwarrior y Skulls encabezando el grupo.

Son cuatro los principales factores detrás del incremento potencial del código malicioso que apunta a los teléfonos inteligentes móviles:

- **Mayor población de dispositivos objetivo:** De acuerdo con Garnter, los *smartphones* representarán 27% de todos los nuevos teléfonos vendidos en el 2009. Los analistas estiman que 80 millones de teléfonos inteligentes se distribuirán en 2006. Los criminales motivados por las ganancias económicas buscan vectores de entrega con muchos usuarios, haciendo a los *smartphones* un objetivo en constante crecimiento.

- **Velocidades de datos más rápidas:** Usando el servicio general de radio de paquetes (GPRS), un archivo de 1MB puede transferirse en aproximadamente 4 minutos. Usando el acceso de paquetes downlink de alta velocidad (HSPDA) 3G, la cantidad de tiempo se reduce a 15 segundos. Las velocidades de datos más rápidas incrementan considerablemente la posibilidad de infección.

- **Popularidad con desarrolladores:** Es muy probable que los cibercriminales exploten las vulnerabilidades de las plataformas de teléfonos más populares.

- **Aumento del poder de cómputo:** Los dispositivos móviles convergentes de hoy corren *software* altamente avanzado. Tienen poder de cómputo equivalente a las PC de escritorio que se vendían hace menos de una década.

Seguridad en Java

Existen varios factores en la seguridad en el entorno Java desde varios puntos de vista:

- **Entorno de ejecución:** La seguridad de un sistema Java para los usuarios está en la máquina virtual, ya que esta es la que controla qué se puede ejecutar y de qué modo, permitiendo controlar qué pueden hacer los programas al ejecutarse en nuestro sistema.

- **Interfaces y arquitecturas de seguridad:** Para que las aplicaciones clientE/Servidor sean seguras, es necesario emplear técnicas criptográficas. Java proporciona una arquitectura en la que integrar estas técnicas y un conjunto de interfaces para simplificar su uso en el desarrollo de aplicaciones.

La seguridad de Java desde un punto de vista teórico considera los siguientes puntos:

- **Criptología:** Estudio de los criptosistemas, sistemas que ofrecen medios seguros de comunicación en los que el emisor oculta o cifra el mensaje antes de transmitirlo para que solo un receptor autorizado (o nadie) pueda descifrarlo
- Técnicas criptográficas
- Certificados digitales
- Protocolos de red seguros

9.6. Mecanismos de protección

Las principales actividades de un sistema operativo son:

1. Protección de los procesos del sistema contra los procesos de usuario
2. Protección de los procesos de usuario contra los de otros procesos de usuario
3. Protección de memoria
4. Protección de los dispositivos

Dominios de protección

Un dominio de protección es un conjunto de pares (objeto, operaciones). Cada par identifica un objeto y las operaciones permitidas sobre él.

En cada instante, cada proceso ejecuta dentro de un dominio de protección. Los procesos pueden cambiar de un dominio a otro en el tiempo; el cómo lo hace depende mucho del sistema. En UNIX, se asocia un dominio a cada usuario+grupo. Dado un usuario y el grupo al cual pertenece, se puede construir una lista de todos los objetos que puede accesar y con qué operaciones. Cuando un usuario ejecuta un programa almacenado en un archivo de propiedad de otro usuario B, el proceso puede ejecutar dentro del dominio de protección de A o B, dependiendo del bit de dominio o *SETUSERID bit* del archivo. Este mecanismo se usa con algunos utilitarios. Por ejemplo, el programa **passwd** debe tener privilegios que un usuario común no tiene, para poder modificar el archivo donde se guardan las claves. Lo que se hace es que el archivo /bin/passwd que contiene el programa es propiedad del superusuario, y tiene el SETUSERID encendido. Este esquema es peligroso: un proceso puede pasar de un estado en que tiene poco poder a otro en que tiene poder absoluto (no hay términos

medios). Cualquier error en un programa como passwd puede significar un gran hoyo en la seguridad del sistema. Cuando se hace una llamada al sistema también se produce un cambio de dominio, puesto que la llamada se ejecuta en modo protegido.

Listas de control de acceso

Alternativamente, podemos guardar la matriz por columnas (descartando las entradas vacías). Es decir, a cada objeto se le asocia una lista de pares (dominio, derechos). Es lo que se conoce como **lista de acceso** o ACL. Si pensamos en archivos de Unix, podemos almacenar esta lista en el nodo-i de cada archivo, y sería algo así como:

((Juan, *, RW), (Pedro, Profes, RW), (*, Profes, R))

En la práctica, se usa un esquema más simple (y menos poderoso), pero que puede considerarse aún una lista de accesos, reducida a 9 bits: 3 para el dueño (RWX), 3 para el grupo, y 3 para el resto del mundo.

Windows NT usa listas de accesos con todo el nivel de detalle que uno quiera: para cualquier usuario o grupo, se puede especificar cualquier subconjunto de derechos para un archivo, de entre {RWXDPO}. .

Capacidades

La otra posibilidad es almacenar la matriz por filas. En este caso, a cada proceso se le asocia una lista de capacidades. Cada capacidad corresponde a un objeto más las operaciones permitidas.

Cuando se usan capacidades, lo usual es que, para efectuar una operación M sobre un objeto O, el proceso ejecute la operación especificando un puntero a la capacidad correspondiente al objeto, en vez de un puntero al objeto. La sola *posesión* de la capacidad por parte del proceso quiere decir que tiene los derechos que en ella se indican. Por lo tanto, obviamente, se debe evitar que los procesos puedan "falsificar" capacidades.

Una posibilidad es mantener las listas de capacidades dentro del sistema operativo, y que los procesos solo manejen punteros a las capacidades, no las capacidades propiamente. Otra posibilidad es cifrar las capacidades con una clave conocida por el sistema, pero no por el usuario. Este enfoque es particularmente adecuado para sistemas distribuidos, y es usado en Amoeba.

Un problema de las capacidades es que puede ser difícil revocar derechos ya entregados. En Amoeba, cada objeto tiene asociado un número al azar, grande, que también está presente en la capacidad. Cuando se presenta una capacidad, ambos números deben coincidir. De esta manera, para revocar los derechos ya otorgados, se cambia el número asociado al objeto. Problema: no se puede revocar selectivamente. Las revocaciones con ACL son más simples y más flexibles.

9.7. Sistemas de confianza

Sistemas de confianza

Ha sido una estrategia de varias compañías entre las que destacan Intel y Microsoft para realizar el control de lo que se permite o no hacer con un ordenador, o, por extensión, cualquier equipo con un microchip que permita esas capacidades.

Básicamente la idea se promociona, como su nombre indica, como un sistema que permite confiar en él para que todo sea más fiable. No obstante, si bien eso podría ser cierto desde el punto de vista del fabricante de *software* o de contenidos, resulta completamente inverso desde el punto de vista del usuario del ordenador.

Esto se debe a que el sistema realiza un control de las actividades del usuario, impidiendo cualquier posibilidad de que haga algo que no le hayan permitido explícitamente las compañías. Por ejemplo, para evitar que se haga una copia de un vídeo de alta resolución que la compañía vendedora haya prohibido en la gestión de derechos digitales, el sistema impedirá que se reproduzca con esa calidad, o incluso por completo (según lo especificado por el vendedor del contenido) si, por ejemplo, existiera una salida no certificada y controlada por dicho sistema.

Modelos formales de sistemas seguros

El modelo de protección puede ser visto abstractamente como una matriz, llamada **matriz de derecho**. Los renglones de la matriz representan dominios y las columnas representan objetos. Cada entrada en la matriz contiene un conjunto de derechos de acceso. Dado que los objetos son definidos explícitamente por la columna, se puede omitir el nombre del objeto en el derecho de acceso. La entrada "*Matriz[i, j]*" define el conjunto de operaciones que un proceso ejecutándose en el dominio "Dj" puede realizar sobre el objeto "Oj".

Considérese la siguiente matriz de acceso:

Dominio \ Objeto	A1	A2	A3	COM1	LPT1
D1	Leer		Leer		
D2				Leer	Imprimir
D3		Leer	Ejecutar		
D4	Leer Escribir		Leer Escribir		

Hay 4 dominios y 5 objetos: 3 Archivos ("A1", "A2", "A3") 1 Puerto Serial y 1 impresora. Cuando un proceso se ejecuta en O1, puede leer los archivos "A1" y "A3".

Un proceso ejecutándose en el dominio "D4" tiene los mismos privilegios que en "D1", pero además puede escribir en los archivos. Nótese que en el puerto serial y la impresora solo pueden ser ejecutados por procesos del dominio "D2".

Seguridad multinivel

Originalmente fue diseñado para manejar seguridad militar.

Hay varios niveles de seguridad:

- Cada objeto (por ejemplo: documento) pertenece a un único nivel.
- Cada persona también pertenece a un único nivel, dependiendo de qué documentos pueda ver.

Reglas que rigen el flujo de información:

- **La propiedad de seguridad simple:** Proceso que se ejecuta en el nivel de seguridad que sólo puede leer objetos de su nivel o de niveles inferiores (se lee hacia abajo).
- **La propiedad *:** Proceso que se ejecuta en el nivel de seguridad que puede escribir sobre objetos de su nivel o de niveles superiores (se escribe hacia arriba).

Si el sistema hace cumplir las dos reglas, se puede demostrar que la información no puede filtrarse a un nivel de seguridad inferior.

Seguridad de libro naranja

El Libro Naranja está dividido en dos partes, la primera de las cuales define las unidades magneto-ópticas (MO), mientras la segunda especifica las unidades CD grabables o CD-R (*CD-Recordable*) o unidades *CD-Write Once.*

Capítulo 12: Diseño de sistemas operativos

12.1. La naturaleza del problema de diseño

El primer problema que se presenta a la hora de diseñar un sistema operativo es definir sus objetivos y especificaciones. Por ejemplo: qué tipo de sistema operativo vamos a diseñar; puede ser sistema de tiempo compartido, monousuario, multiusuario, distribuido, de tiempo, etc.

Es importante cumplir tanto los requerimientos de los usuarios del sistema así como también los requerimientos o metas del propio sistema operativo.

Metas

Los diseñadores deben tener una idea clara de lo que quieren al diseñar un sistema operativo. Existen cuatro objetivos esenciales en el diseño de un sistema operativo de propósito general:

- Definir abstracciones
- Proporcionar operaciones primitivas
- Garantizar el aislamiento
- Administrar el *hardware*

¿Por qué es difícil diseñar sistemas operativos?

- Por la inercia y el deseo de mantener la compatibilidad con sistemas anteriores.
- Por no ajustarse a los buenos principios de diseño.
- Por la complejidad y tamaño de los sistemas operativos actuales.
- Por la concurrencia de usuarios, de dispositivos de E/S, de procesos, etc.
- Por la necesidad de compartir archivos y recursos de forma controlada por el usuario.
- Por la portabilidad de los sistemas operativos.

12.2. Diseño de interfaces

Un sistema operativo presta un conjunto de servicios que conforman las interfaces que interactúan entre el usuario y el sistema; por ejemplo, un controlador de dispositivos.

Principios orientadores

El diseño de interfaces debe ser sencillo, completo y susceptible de implementar fácilmente. Los principios son:

1. **Sencillez:** Fácil de entender e implementar

2. **Integridad:** Que haga lo que el usuario desea hacer

3. **Eficiencia:** En la implementación

Paradigmas de ejecución

Existen dos:

1. **Algorítmico:** Se basa en la idea de que un programa se inicia para realizar alguna función que conoce anticipadamente u obtiene sus parámetros.

2. **Controlado por sucesos:** Habilita un servicio en espera de que el sistema operativo le notifique algún suceso para iniciar su actuación.

Paradigma de datos

Se refiere a la forma de tratamiento que los sistemas operativos dan a los dispositivos de almacenamientos y las estructuras del sistema, y cómo se presentan al programador; desde cintas secuenciales, tarjetas perforadas, archivos en disco y las impresoras tratadas como unidades de E/S.

La interfaz de llamadas al sistema

El sistema operativo es una interfaz que oculta las peculiaridades del *hardware*. Para ello, ofrece una serie de servicios que constituyen una máquina virtual más fácil de usar que el *hardware* básico. Estos servicios se solicitan mediante llamadas al sistema.

La forma en que se realiza una llamada al sistema consiste en colocar una serie de parámetros en un lugar específico (como los registros del

procesador), para después ejecutar una instrucción del lenguaje máquina del procesador denominada ***trap*** (en castellano, trampa). La ejecución de esta instrucción máquina hace que el *hardware* guarde el contador de programa y la palabra de estado del procesador (PSW, *Processor Status Word*) en un lugar seguro de la memoria, cargándose un nuevo contador de programa y una nueva PSW. Este nuevo contador de programa contiene una dirección de memoria donde reside una parte (un programa) del sistema operativo, el cual se encarga de llevar a cabo el servicio solicitado. Cuando el sistema operativo finaliza el servicio, coloca un código de estado en un registro para indicar si hubo éxito o fracaso, y ejecuta una instrucción *return from trap*. Esta instrucción provoca que el *hardware* restituya el contador de programa y la PSW del programa que realizó la llamada al sistema, prosiguiéndose así su ejecución.

Normalmente, los lenguajes de alto nivel tienen una (o varias) rutinas de biblioteca por cada llamada al sistema. Dentro de estos procedimientos se aísla el código (normalmente en ensamblador) correspondiente a la carga de registros con parámetros, a la instrucción *trap*, y a obtener el código de estado a partir de un registro. La finalidad de estos procedimientos de biblioteca es ocultar los detalles de la llamada al sistema, ofreciendo una interfaz de llamada al procedimiento. Como una llamada al sistema depende del *hardware* (por ejemplo, del tipo de registros del procesador), la utilización de rutinas de biblioteca hace el código portable.

El número y tipo de llamadas al sistema varía de un sistema operativo a otro. Existen, por lo general, llamadas al sistema para ejecutar ficheros que contienen programas, pedir más memoria dinámica para un programa, realizar labores de E/S (como la lectura de un carácter de un terminal), crear un directorio, etc. Ejemplos de rutinas de biblioteca que realizan llamadas al sistema en un entorno del sistema operativo C-UNIX son: *read, write, malloc, exec,* etc.

Principios de diseño:

Principio 1. Sencillez
Las interfaces sencillas son más fáciles de entender e implementar.

Principio 2. Integridad
La interfaz debe permitir hacer todo lo que los usuarios necesitan hacer. Pero los mecanismos que soportan la interfaz deben ser pocos y sencillos (deben hacer una única cosa, pero deben hacerla bien).

Principio 3. Eficiencia
La implementación de los mecanismos debe ser eficiente. Debe ser intuitivamente obvio para el programador cuánto cuesta una llamada al sistema.

12.3. Implementación

Estructura del sistema

1. **Cargador:** Cualquier programa que requiere ser ejecutado en la computadora, deberá ser transferido desde su lugar de residencia a la memoria principal.

2. **Cargador para el sistema operativo:** Este programa se encarga de transferir desde algún medio de almacenamiento externo (disco, cinta o tambor) a la memoria principal, los programas del sistema operativo que tienen como finalidad establecer el ambiente de trabajo del equipo de cómputo. Existe un programa especial almacenado en memoria ROM que se encarga de accesar a este programa cargador. Cuando el sistema operativo está cargado en memoria, toma el control absoluto de las operaciones del sistema.

3. **Cargador incluido en el sistema operativo**: Su función es cargar a memoria todos los archivos necesarios para la ejecución de un proceso.

4. **Supervisor (ejecutivo o monitor):** Es el administrador del sistema que controla todo el proceso de la información por medio de un gran número de rutinas que entran en acción cuando son requeridos. Funge como enlace entre los programas del usuario y todas las rutinas que controlan los recursos requeridos por el programa para posteriormente continuar con su ejecución.

   El supervisor también realiza otras funciones como son:

   - Administración de la memoria
   - Administración de las rutinas que controlan el funcionamiento de los recursos de la computadora
   - Manejo de Archivos
   - Administración y control de la ejecución de los programas

5. **Lenguaje de comunicación:** Es el medio a través del cual el usuario interactúa directamente con el sistema operativo. Está formado por comandos que son introducidos a través de algún dispositivo. Generalmente, un comando consta de dos partes, la primera formada por una palabra que identifica el comando y la acción por realizar, y la segunda parte está constituida por un conjunto de valores o parámetros que permiten seleccionar diversas operaciones de entre los que dispone el comando.

6. **Utilería de sistema:** Son programas o rutinas del sistema operativo que realizan diversas funciones de uso común o aplicación frecuente, como son: clasificar, copiar e imprimir información.

Estructura del sistema operativo:

- Hay varias alternativas: por capas, *exokernel*s, cliente-servidor, etc.

- Partes del sistema operativo deben facilitar la construcción de otras partes del sistema operativo:
    - Ocultar interrupciones.
    - Proporcionar mecanismos sencillos de concurrencia.
    - Posibilitar la construcción de estructuras de datos dinámicas,
    - Etc.

- ¡Prestar especial atención a los manejadores de dispositivo!
    - Constituyen una parte muy importante del sistema global.
    - Son la principal fuente de inestabilidad.

Mecanismos y políticas

- La separación entre mecanismos y políticas ayuda a la coherencia arquitectónica y a la estructuración del sistema

- Los mecanismos se pueden implementar en el núcleo y las políticas fuera o dentro del núcleo

- **Ejemplo 1. Planificación de procesos**
    - **Mecanismo:** Colas multinivel por prioridad donde el planificador siempre selecciona al proceso listo de mayor prioridad
    - Política: planificación apropiativa o no, asignación de prioridades a procesos por usuario, etc.

- **Ejemplo 2. Gestión de la memoria virtual**
  - **Mecanismo:** Administración de la MMU, listas de páginas ocupadas y libres, transferencia de páginas entre memoria y disco
  - **Política:** Reemplazo de páginas global o local, algoritmo de reemplazo de páginas, etc.

Ortogonalidad

- Capacidad de combinar conceptos distintos de forma independiente. Es consecuencia directa de los principios de sencillez e integridad.

- Ejemplo 1. Procesos e hilos: Separan la unidad de asignación de recursos (proceso) de la unidad de planificación y ejecución (hilo), permitiendo tener varios hilos dentro de un mismo proceso.

- Ejemplo 2. Creación de procesos en Unix: Se diferencia entre crear el proceso en sí (fork) y ejecutar un nuevo programa (exec), lo que permite heredar y manipular descs. de fichero.

- En general, tener un número reducido de elementos ortogonales que pueden combinarse de muchas maneras da pie a un sistema pequeño, sencillo y elegante.

Asignación de nombres

- Casi todas las estructuras de datos duraderas que utiliza un sistema operativo tienen algún tipo de nombre o identificador (nombre de dispositivo, de fichero, identificador de proceso, etc.).

- Es común que los nombres se asignen a dos niveles:
  - **Externo:** Cadenas de caracteres (estructuradas o no) que usan los usuarios.
  - **Interno:** Identificación usada internamente por el sistema operativo.

- Debe existir algún mecanismo que permita asociar unos nombres con otros. Ejemplo: los directorios (enlazan nombres de ficheros con nodos-i).

- Un buen diseño debe estudiar con detenimiento cuántos espacios de nombres van a necesitarse, qué sintaxis tendrán los nombres, cómo van a distinguirse, etc.

Estructuras estáticas o dinámicas

- ¿Las estructuras de datos del sistema operativo deben ser estáticas o dinámicas?

- Las estáticas son más comprensibles, más fáciles de programar y de uso más rápido.

- Las dinámicas son más flexibles y permiten adaptarse a la cantidad de recursos disponibles. Problema: necesitan un gestor de memoria dentro del propio sistema operativo.

- Según el caso, puede ser más adecuado un tipo u otro.
    - Ejemplo:
        - Pila de un proceso en el espacio de usuario: estructura dinámica
        - Pila de un proceso en el espacio de núcleo: estructura estática

- También son posibles estructuras pseudo – dinámicas

Técnicas útiles

- Ocultación del *hardware*

    - Ocultar las interrupciones, convirtiéndolas en operaciones de sincronización entre hilos (unlock en un mutex, envío de un mensaje, etc.).
    - Ocultar las peculiaridades de la arquitectura *hardware* si se desea facilitar la transportabilidad del sistema operativo:
        - Los fuentes del sistema operativo deben ser únicos.
        - La compilación debe ser condicional.
        - La parte de código dependiente debe ser lo más pequeña posible.
    - Ejemplo 1: HAL (*Hardware* Abstraction Layer) de Windows
    - Ejemplo 2: directorios arch e include/asm-* en Linux

- Indirección

    - ¡Flexibilidad!
    - Ejemplo: entrada por teclado. Al pulsar una tecla se obtiene un valor que no se corresponde con su código ASCII
      Posibilidad de utilizar configuraciones distintas de teclados

- Ejemplo: salida por pantalla. El código ASCII es el índice de una tabla con el patrón de bits del carácter por representar
  Posibilidad de configurar el tipo de letra de pantalla
- Ejemplo: nombres de dispositivo. En Linux, los nombres de los ficheros especiales son independientes de los números mayor y menor de dispositivo.
  Posibilidad de cambiar el nombre a un dispositivo
- Reentrabilidad
  - Se refiere a la capacidad de un fragmento de código dado para ejecutarse dos o más veces de forma simultánea
  - La ejecución simultánea se puede dar en varios casos:
    - En un multiprocesador dos o más procesadores pueden estar ejecutando la misma porción de código
    - En un monoprocesador puede llegar una interrupción que ejecute la misma porción de código que se estaba ejecutando.
  - Lo mejor, incluso en un monoprocesador, es que:
    - la mayor parte del sistema operativo sea reentrante (para que no se pierdan interrupciones),
    - que las secciones críticas se protejan
    - que las interrupciones se inhabiliten cuando no se puedan tolerar
- Fuerza bruta
  - ¡Optimizar cuando realmente merezca la pena!
  - Ejemplo 1. ¿Merece la pena tener ordenada una tabla de 100 elementos que cambia continuamente para que las búsquedas sean más rápidas?
  - Ejemplo 2. ¿Merece la pena optimizar una llamada al sistema que tarda 10 ms para que tarde 1 ms si se utiliza una vez cada 10 segundos?
  - ¡Optimizar las funciones y estructuras de datos de la ruta crítica! Ejemplo: cambio de contexto

12.4. Desempeño

- ¿Qué es mejor, un sistema rápido y poco fiable, o uno lento pero fiable?

- Las optimizaciones complejas suelen llevar a errores.
  Optimizar sólo si realmente es necesario.

- ¿Qué es mejor, un sistema operativo sencillo y rápido, o uno complejo y lento? ¡Lo pequeño es bello! Antes de añadir una funcionalidad nueva comprobar que realmente merece la pena.

- En cualquier caso, antes de optimizar, tener en cuenta que lo bastante bueno es bastante bueno.

- Otra consideración importante es el lenguaje de programación por utilizar.

Uso de cachés

- Se aplican en situaciones en las que es probable que el mismo resultado se necesite varias veces.
- Son especialmente útiles para dispositivos de E/S.
- Ejemplo 1. Caché de bloques o caché de disco
- Ejemplo 2. Caché de entradas de directorio
- Ejemplo 3. Caché de páginas

Optimización del caso común

- En muchos casos es recomendable distinguir entre el caso más común y el peor caso posible:
    - Es importante que el caso común sea rápido.
    - El peor caso, si no se presenta a menudo, sólo tiene que manejarse correctamente.
- Ejemplo: llamada EnterCriticalSection del API Win32
    - Algoritmo:
        - Comprobar y cerrar mutex en espacio de usuario
        - En caso de éxito (Entrar a la sección crítica (no ha hecho falta una llamada al sistema)
        - En caso de fracaso) Ejecutar una llamada al sistema que bloquee al proceso en un semáforo hasta que pueda entrar a la sección crítica
    - Si lo normal es que la primera comprobación tenga éxito, nos habremos ahorrado entrar al núcleo del SO.

12.5. Administración de proyectos

El mes-hombre mítico

- Según Brooks: Un programador sólo puede producir 1000 líneas de código depurado al año en un proyecto grande
    - Los proyectos grandes, con cientos de programadores ≠ proyectos pequeños ☐ Resultados no extrapolables
    - En proyectos grandes, el trabajo se compone de:
    - 1/3 planificación
    - 1/6 codificación
    - 1/4 prueba de módulos
    - 1/4 prueba del sistema

- Para Brooks, no existe una unidad hombre–mes: si 15 personas tardan dos años en un proyecto, 360 no van a tardar 1 mes. Razones:
    - El trabajo no puede dividirse totalmente en actividades paralelas
    - El trabajo debe dividirse en muchos módulos y las interacciones entre ellos crecen con el cuadrado del no de los mismos
    - La depuración es secuencial

- Ley de Brooks: ((la adición de personal a un proyecto de *software* atrasado lo atrasa más aún))

12.6. Tendencias en el diseño de sistemas operativos

Sistemas operativos con espacio de direcciones grandes

Con las expansiones de las direcciones de 32 a 64 bits se podrían presentar algunas conveniencias:

- Eliminarse el concepto de sistema de archivos y colocarlos, conceptualmente, en la memoria virtual de la máquina.

- Crear objetos en el espacio de direcciones y eliminar sus referencias hasta que se eliminen.

- Con la persistencia de los objetos en el espacio de direcciones, sería factible que múltiples procesos se ejecuten en ese mismo espacio de direcciones paralelamente.

Redes

Se prevé que las redes sean la base de los futuros sistemas operativos. Por consiguiente, el termino datos locales y datos remotos tendería a desaparecer, y los exploradores serían, tal como ahora, un explorador de todo el contenido de la red, pero inmerso en el mismo sistema operativo.

Sistemas paralelos y distribuidos

El paralelismo de los sistemas operativos se ve proyectado a trabajar sobre varios procesadores en un mismo computador. Esto es, sistemas operativos que gestionen más de un CPU para el procesamiento de la máquina. Esto ya es una realidad.

Multimedia

La fusión de varios aparatos electrónicos actuales dará la oportunidad a los sistemas operativos para que trabajen para conjuntar su funcionalidad. Entonces se podrá ver a los sistemas operativos gestionar audio, video y gráficas en electrodomésticos tales como los televisores o equipos de sonido.

Sistemas incrustados

Como una norma común, la totalidad de los aparatos electrónicos que se utilizan hoy día traen implementados internamente su propio sistema operativo. Es un hecho que las lavadoras, microondas, televisores y otros artefactos hagan uso de su propio sistema operativo para funcionar adecuadamente.

Referencias

Martínez Rodríguez, José David y Hervás, Alberto Sanjuán (1998). *Material digital del curso Sistemas operativos*. España, Universidad de Jaén. Extraído el 28 de noviembre de 2008, de:
wwwdi.ujaen.es/~lina/TemasSO/MEMORIAVIRTUAL/4ReemplazodePaginas.htm

s. a., s. f. *Dispositivos de entrada / salida*. Extraído el 28 de noviembre del 2008.
www.elo.utfsm.cl/~elo321/01-2009/docs/dispositivos.pdf

Tanenbaum, Andrew S. (2003). *Sistemas operativos modernos*. Pearson Educación. Segunda Edición. México. Páginas 976.

Trejo Ramírez, Ricardo (26 de junio del 2002). *Gestión de memoria*. Extraído el 28 de noviembre del 2008 de
www.monografias.com/trabajos10/gesmem/gesmem.shtml