SISTEMAS OPERATIVOS

UNA VISIÓN GENERAL

Miguel Ángel Fernández Marín
Mateo Gerónimo Lezcano Brito
Zoila Zenaida García

Diseño de carátula: Dr.C. Liéter Elena Lamí Rodríguez del Rey

Edición: Dr.C. Liéter Elena Lamí Rodríguez del Rey

Corrección: MSc. Alicia Martínez León

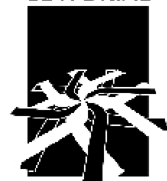
Dirección editorial: Dr. C. Jorge Luis León González

Sobre la presente edición:

© Editorial Universo Sur, 2021

ISBN: 978-959-257-628-5

Podrá reproducirse, de forma parcial o total, siempre que se haga de forma literal y se mencione la fuente.



Editorial: "Universo Sur".

Universidad de Cienfuegos. Carretera a Rodas, Km 3 □.

Cuatro Caminos. Cienfuegos. Cuba.

CP: 59430

SISTEMAS OPERATIVOS
UNA VISIÓN GENERAL

Miguel Ángel Fernández Marín
Mateo Gerónimo Lezcano Brito
Zoila Zenaida García

SISTEMAS OPERATIVOS

UNA VISIÓN GENERAL

Miguel Ángel Fernández Marín
Mateo Gerónimo Lezcano Brito
Zoila Zenaida García

Introducción

Los sistemas operativos (SO) son programas que actúan como una interfaz entre el sistema de cómputo y los programas de aplicación. Su principal función es asignar y controlar los recursos que necesitan los programas, los cuales pueden ser:

- Recursos de *hardware*: la memoria interna; el procesador central (*Central Process Unit-CPU*); equipos de entrada o de salida: el ratón (*mouse*), las impresoras, el teclado y el monitor, entre otros; equipos de almacenamiento externo: discos de diferentes tipos, memorias *USB*, cintas magnéticas, entre otros (Lezcano Brito, 2018).
- Recursos de *software*: la tabla de archivos abiertos, las estructuras de datos que controlan el uso de la memoria o del espacio en los equipos de almacenamiento externo, entre otros.
- Debido al avance en el desarrollo del *hardware*, muchas de las funciones originales de los SO han migrado hacia los equipos correspondientes, por ejemplo, el SO puede realizar las solicitudes de entrada/salida (E/S) a los discos en forma genérica debido a que estos tienen un procesador propio para manipular esas peticiones, lo que libera al SO de conocer las particularidades propias de cada tipo de disco.

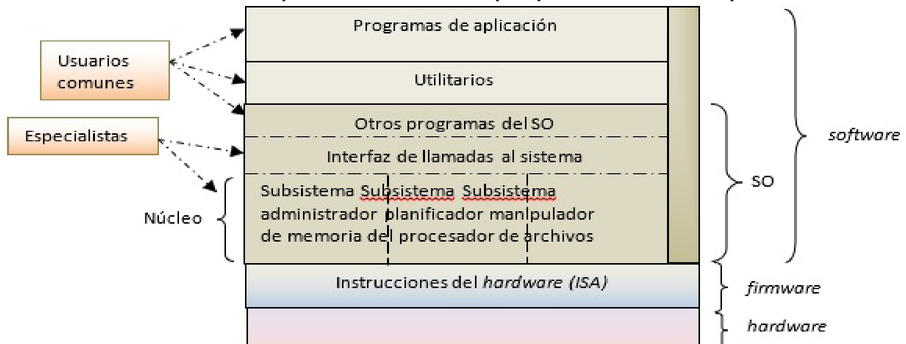


Figura 1. Vista de capas jerárquicas en un sistema de cómputo.

Es usual ver el *hardware* y el *software* como un conjunto de capas jerárquicas, la figura 1 muestra una concepción esquematizada de estas capas. Obsérvese que los usuarios, generalmente, no actúan con el *hardware* debido a que el SO los aísla de las particularidades que caracterizan a esos medios.

La mayoría de los usuarios interactúan con los programas de aplicación,

otro grupo más reducido utiliza un conjunto de programas utilitarios, muchos de los cuales se distribuyen con el SO, aunque no forman parte de él. Un grupo aún más pequeño actúa con las partes del SO que no están contenidas en el núcleo, lo que incluye la interfaz que brinda el SO para acceder a sus funciones básicas (conocida como interfaz de llamadas al sistema); por último, los desarrolladores del SO actúan con todo el SO, incluyendo los módulos de su núcleo (*kernel*).

El *kernel* no es más que el que concentra todas las funcionalidades básicas del sistema operativo: gestión de los procesos, manipulación del sistema de archivos, control de los dispositivos de *hardware*, gestión de memoria, entre otras (Millo Sánchez, et al., 2016).

Cuando se desarrollan aplicaciones se usan, generalmente, diversos lenguajes de programación de alto nivel y se deja al SO la responsabilidad de manejar el h*ardware*, pero cuando es necesario trabajar directamente con los componentes físicos es necesario usar el conjunto de instrucciones del *hardware* (*ISA- Instruction Set Architecture*), esta última tarea tiene un alto grado de complejidad y debería analizarse la necesidad de hacerla porque posiblemente el SO proporcione una manera cómoda y más abstracta de realizarla.

La interfaz de llamadas al sistema, permite el acceso a las funciones del núcleo del SO y puede existir una biblioteca, nombrada interfaz para programas de aplicación (*API- Application Programming Interface*), que ofrece múltiples servicios de acceso a los recursos a través de un lenguaje de alto nivel, garantizando la compatibilidad de los programas que se hagan siguiendo su estándar, por ejemplo, las bibliotecas de hilos *Pthreads* y *Win32* de los SO tipo Unix y Windows respectivamente.

Área de servicios que ofrece el SO

Los SO brindan diferentes facilidades, algunas de ellas responden a características particulares de una implementación dada, pero se pueden distinguir los siguientes servicios generales:

1. De apoyo a la programación: Se brindan a través de diversos utilitarios para editar y poner a punto programas, ellos en sí no forman parte del SO pero se distribuyen junto con él.

2. De ejecución de programas: Para ejecutar un programa es necesario cargar su código y sus datos en la memoria principal, además de inicializar algunos archivos y periféricos, todo lo cual es responsabilidad del SO.

3. De acceso a los equipos de entrada/salida: Hoy en día existe una diversidad notable de equipos de E/S que tienen disímiles y específicas instrucciones para controlar su operación. El SO debe proporcionar una interfaz uniforme que permita a los usuarios manejar los equipos abstrayéndose de esas particularidades.

4. De control de acceso a los archivos: El SO debe conocer la naturaleza de los equipos así como las estructuras de datos que permiten acceder a los contenidos de esos medios de almacenamiento, por eso se hace necesario que existan mecanismos de protección que controlen el acceso a ellos.

5. De acceso al sistema: Es un conjunto de funciones que controlan quiénes y cómo pueden acceder a los recursos (incluidos los datos), adicionalmente deben resolver diversos conflictos que pueden surgir cuando se comparten dichos recursos.

6. De detección de errores: Cuando un sistema de cómputo está en acción pueden surgir múltiples errores, los cuales pueden ser provocados por el *hardware* o el *software*; el SO es responsable de detectarlos y tomar la decisión adecuada en cada caso.

7. De estadística: Se hace necesario que el SO lleve ciertas estadísticas referidas al uso de los recursos debido a que esa información resulta útil para garantizar un mejor rendimiento y usar políticas adecuadas para manejar los medios.

Los SO usan un conjunto de instrucciones, en lenguaje de máquina, definido por la arquitectura del equipo de cómputo sobre el que se instalan que, habitualmente, se conoce por sus siglas en inglés ISA.

El conjunto ISA establece la frontera entre el *hardware* y el *software*, las instrucciones a este nivel pueden terminarse en uno o varios ciclos de datos y utilizan los registros del procesador y otros recursos de hardware.

Para programar al nivel del ISA, se utiliza el lenguaje de máquina o el

ensamblador (un nivel más abstracto). En ese caso las herramientas

La evolución de los SO

Todo profesional debe conocer la historia de su profesión y aunque este es un argumento loable, no es el único que obliga a realizar este brevísimo relato. Sin lugar a dudas el estudio de la evolución de los SO justifica muchas de las decisiones que han tomado sus diseñadores, las cuales han tenido en cuenta las posibilidades que brinda el desarrollo del hardware de cada época.

Sin el auxilio del SO

Solo los programadores usaban las primeras computadoras (surgidas entre 1940 y 1950), ellos tenían que interactuar directamente con el *hardware* debido a que no existían los SO. Toda la memoria estaba disponible para ellos y podían usarla sin ningún tipo de restricción.

Para programar se usaba el lenguaje de máquina, que es particular para cada tipo de equipo; una vez hecho el programa, se cargaba en memoria usando un equipo de entrada. Por ejemplo, un lector de cinta de papel o de tarjetas perforadas. Existían y existen dos posibilidades cuando se intenta ejecutar un programa.

- Que se encuentren errores: En las computadoras de esa época se indicaba por luces en un panel y el programador debía estar atento a esas indicaciones para detener el procesamiento, corregir los errores y reiniciar el ciclo de carga y ejecución. Para poder encontrar los errores, muchas veces era necesario hacer un vaciado de memoria, que es una copia del contenido de la memoria principal, en código de máquina (binario, octal o hexadecimal). Hoy en día no se usa esta técnica debido a que existen lenguajes de alto nivel y programas de puesta a punto (*debuggers*) que significa un proceso que comprende las etapas de edición, compilación, corrección de errores, ejecución y prueba interactivos muy poderosos y versátiles (Lezcano Brito, 2018).
- Que no existan errores: En este caso el programa terminaba y sus salidas quedaban plasmadas en una impresora.

Esta concepción inicial tenía dos problemas dignos de destacar:

- El primero es la forma en que se planificaba el uso de la computadora, que simplemente consistía en reservar un tiempo para utilizarla

(conocido como tiempo de máquina). Es fácil apreciar que la estimación del tiempo podía crear un conflicto lo mismo si se sobreestimaba que si se estimaba por debajo de la necesidad real.

- En el primer caso se malgastaba ese tiempo y en el segundo caso era necesario negociar con las personas que tenían trabajos planificados para esa hora.
- El tiempo de operación: Un programa simple, denominado trabajo (*job*) o tarea (*task*) en esa época, podía constar de varias etapas; por ejemplo: cargar el compilador y el programa (programa fuente) en la memoria, compilar el programa o el conjunto de programas que conformaban la aplicación para obtener como salida el o los programas objetos, enlazar los distintos módulos objetos obtenidos en el paso anterior para producir un programa ejecutable y por último cargar el ejecutable en memoria para ejecutarlo.

Cualquiera de los pasos descritos y tal vez otros más, podían provocar fallos que muchas veces exigían, entre otras cosas, montar y desmontar cintas magnéticas, proveer a los lectores de sus entradas, obligando a iniciar todo el proceso de nuevo.

El monitor residente y los sistemas por lotes (batch)

Las primeras computadoras eran muy costosas y por ese motivo era muy importante tratar de maximizar el uso del procesador, lo que no se lograba con el proceder descrito en el análisis anterior. Los programadores se percataron que muchas tareas eran comunes a todos los trabajos y decidieron automatizarlas creando el monitor residente.

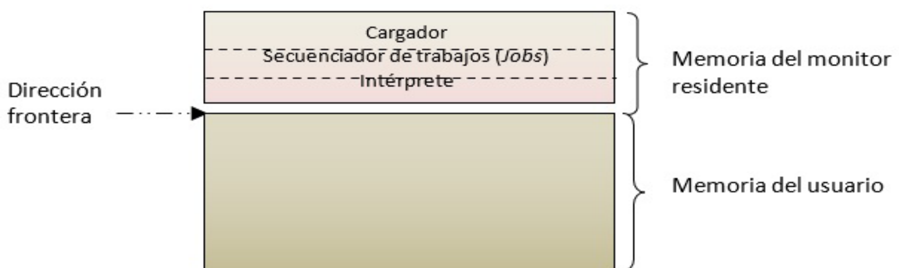


Figura 2. Distribución de la memoria en el monitor residente.
Fiuente: Lezcano Brito (2018).

El esquema del monitor residente se regía por el mismo principio de cargar (Lezcano Brito, 2018) un único programa (completo) en memoria y en forma contigua, pero en este caso la memoria se dividía en dos áreas:

- En la primera se cargaba el monitor residente. Ese código residía permanentemente en memoria (de ahí su nombre) y tenía la tarea de monitorear (controlar) lo que se hacía.
- La segunda área se asignaba al usuario. Esta implementación se usó por primera vez en el sistema *Fortran Monitoring System* (Lezcano Brito, 2017).

El código del monitor residente no se podía alterar, por eso se usaba una dirección que sirviera de frontera entre su zona y la del usuario. Cada dirección generada se verificaba para tener certeza de que no se violara la frontera (Lezcano Brito, 2017, 2018); si había una violación, el monitor tomaba la decisión adecuada (podía ser tan drástica como abortar el proceso violador).

Obsérvese que ahora las direcciones de los programas de usuario no comienzan en la dirección 0 sino en la dirección frontera. Si esta última no cambia nunca, el compilador puede generar direcciones absolutas, a partir de la dirección frontera, pero si es variable existirá la necesidad de relocalizar las direcciones de los programas de usuarios que significa calcular su dirección real en la memoria física de la computadora.

Con el monitor residente surgió el trabajo denominado operador de computadoras, a partir de ese momento los usuarios entregaban sus programas al operador (sobre tarjetas perforadas) y este último formaba un lote con todos, agregándole un conjunto de órdenes (al inicio y al final del lote) y algunas marcas para separarlos (al inicio y al final de cada uno). Después de lo anterior introducía el lote (el conjunto) en un equipo de entrada y le daba el control al monitor, el cual debía realizar todo el trabajo que se hacía manualmente en el esquema analizado anteriormente.

Las órdenes que agregaba el operador de computadoras se especificaban en un lenguaje de control de tareas (*JCL, Job Control Language*) y estaban dirigidas al monitor, el cual usaba:

- Un intérprete para entender las órdenes recibidas a través del programa JCL.
- Un cargador para cargar los trabajos de acuerdo al orden establecido por el secuenciador (Figura 2). Algunas órdenes en JCL se aprecian a continuación (el símbolo // se usa para hacer comentarios, que son ignorados durante el proceso de interpretación o traducción):

```
$JOB   //Inicio del programa de usuario.
$FTN   //Ejecutar el compilador de FORTRAN.
$ASM   // Ejecutar el ensamblador.
$RUN   // Ejecutar el programa de usuario.
$END   //Fin del programa de usuario.
```

En esta solución existe una sobrecarga que está relacionada con los cambios que deben hacerse para cambiar desde el modo de operación del monitor (que actúa sobre un área de memoria protegida) hacia el modo de operación de los programas y viceversa, pero esa sobrecarga es necesaria, aún continúa hoy en día y se conoce como modo de operación dual: modo núcleo (*kernel*) - modo usuario.

Se puede considerar al monitor residente como la génesis de los SO porque su función era automatizar el trabajo del operador de computadora. Debe decirse que los operadores de computadoras de esa época eran personas muy especializadas en operar un tipo muy particular de máquina (todas tenían distintos modos de operación).

Sistemas operativos multiprogramados de tratamiento por lote (batch)

A pesar de que la solución anterior ayudó a disminuir el tiempo de inactividad de las computadoras, aún existían períodos grandes de espera que estaban relacionados con la baja velocidad de los equipos de E/S comparada con la velocidad del procesador.

Se notó que se podía cargar más de un programa de usuario; por ejemplo, un programa A y otro B, para después comenzar el procesamiento por uno de ellos, e intercambiar el procesador, el conjunto de trabajos se le denominó lote.

El proceder era el siguiente: el sistema asignaba el procesador a uno de los trabajos; por ejemplo, el A, pero ahora cuando A hacía una ope-

ración de E/S no se esperaba por su terminación, sino que se le retiraba el procesador para asignárselo a B. Esta idea tomó en cuenta que los equipos de E/S siempre han sido muy lentos con relación al procesador.

Esta manera de trabajar con más de un trabajo a la vez se conoce como multiprogramación (*multiprogramming*) o multitarea (*multitasking*) y es un tema central de los SO modernos que se abordará en el capítulo I.

Existieron dos implementaciones de los sistemas por lote: fuera de línea y en línea.

Operación fuera de línea (off-line operation)

El surgimiento de los equipos basados en cintas magnéticas, muchos más veloces que los lectores de tarjetas perforadas, hizo pensar en la posibilidad de disminuir las demoras producidas por los lentos lectores de tarjetas y surgió un proceder que se denominó operación fuera de línea (*off-line*).

La solución usaba dos computadoras satélites especializadas y más baratas que la computadora principal:

- Una máquina satélite S1, dedicada a las entradas, tenía acoplado un lector de tarjetas desde el cual se cargaba el lote hacia una cinta magnética. Una vez llenada la cinta, el operador la desmontaba para después montarla en la máquina principal y dar la orden de cargar todo el lote. La máquina principal hacia su trabajo y enviaba los resultados a otra cinta magnética.
- Cuando la máquina principal terminaba el procesamiento del lote, el operador desmotaba la cinta con las salidas y la llevaba a una segunda máquina satélite S2, que tenía una impresora acoplada, desde la cual se imprimían todas las salidas.

Operación en línea (on-line operation)

Con el tiempo los discos sustituyeron las cintas y las tarjetas leídas pasaron a almacenarse en un disco, acoplado directamente a la computadora principal; la localización de cada trabajo se especificaba en una tabla creada y mantenida por el SO. Ahora el SO tomaba las entradas directamente del disco, no en forma secuencial como la cinta sino accediendo directamente a las localizaciones deseadas.

La misma estrategia se aplicaba a la salida que también se almacena-

ba en el disco hasta completarla. Una vez que el trabajo terminaba de ejecutarse se podía imprimir su salida (ya estaba completa).

Este tipo de sistema recibió el nombre de SPOOL (*Simultaneous Peripheral Operation On Line*). Hoy en día algunos SO proveen el servicio spooler para controlar la cola de impresión.

Sistemas de tiempo compartido (time-sharing)

Con el tiempo, los sistemas operativos por lotes se volvieran ineficientes, debido a que los usuarios de los programas no podían interactuar con ellos y esa era una necesidad que muchos deseaban.

Aunque la posibilidad de interactuar con los programas hoy en día es algo corriente, esa no era la panorámica que presentaban los sistemas operativos de la década de 1960 y por eso surgió el concepto de compartir el tiempo. La idea era ejecutar cada trabajo por un intervalo de tiempo definido y después de vencido ese tiempo quitarle el procesador para entregárselo a otro, con el cual se seguiría la misma política de desalojo; dando la idea de que todos los trabajos se ejecutaban a la vez.

En esta concepción, el SO sigue la siguiente rutina:

- Asigna el procesador a un proceso por un pequeño tiempo, denominado *quantum*. El proceso comienza su ejecución.
- Cuando termina el *quantum*, desaloja al proceso del procesador, lo envía a una cola de espera y comienza el ciclo de nuevo por el paso 1.

Obsérvese que los sistemas *batch* y los de tiempo compartido son multiprogramados, pero tienen las siguientes diferencias:

- Con relación al objetivo central: Los sistemas por lotes tratan de maximizar el uso del procesador, mientras los sistemas de tiempo compartido se trazan la meta de minimizar el tiempo de respuesta de los procesos o sea enviar salidas a los usuarios en cuanto estén y no esperar a que finalice el lote.
- La forma de dar las órdenes: En los sistemas por lotes se hace a través de un conjunto de directivas que se incorporan al lote, mientras en los sistemas de tiempo compartido el usuario utiliza diversos comandos para interactuar con el trabajo desde una terminal.

Los sistemas multiprogramados agregaron una nueva complejidad al SO

debido a que era necesario proteger el espacio de direcciones de los trabajos cargados en memoria para que no se hicieran daños entre ellos, también era necesario proteger el acceso a los archivos.

Un ejemplo clásico de SO de tiempo compartido de esa época fue el CTSS (*Compatible Time-Sharing System*) que fue desarrollado en el MIT para una máquina IBM 709 y después se implementó también para la IBM 7094.

Un SO es un *software* extremadamente complejo que debe manejar una cantidad apreciable de recursos, cada uno de ellos con diversas especificaciones. Las investigaciones en este campo han permitido resolver esos problemas para presentar una interfaz abstracta a los usuarios, de manera que sea fácil interactuar con cada uno de los medios; por ejemplo cuando un usuario utiliza un archivo no se preocupa por las características físicas del equipo sobre el que está almacenado, sin embargo esa diferencia es sustancial para el SO que tiene que lidiar con las características específicas de cada medio (aunque hoy en día muchos de ellos tienen manipuladores propios), el capítulo III ofrece una visión general de los equipos de almacenamiento externo y el capítulo IV se dedica al sistema de archivos.

Los recursos que, en general, debe manejar el SO son: El procesador, la memoria principal, los equipos de entrada/salida y los equipos de almacenamiento masivo (la memoria externa o secundaria).

Históricamente se han usado las palabras trabajo (*job*) y tarea (*task*) para referirse a los programas en ejecución, pero actualmente se usa la palabra proceso, donde el término se usó por primera vez por los diseñadores del SO Multics en 1960.

Definición. Un proceso es un programa en ejecución.

Por ahora puede aceptarse esta definición sencilla a la cual se incorporarán más elementos próximamente. Obsérvese que aquí se toma distancia del concepto de programa, debido a que los procesos son entes activos que realizan acciones y poseen diversos recursos (para ejecutar necesitan: una cierta cantidad de memoria, el procesador y quizás otros recursos).

La historia de los SO es más rica de lo descrito en los párrafos pre-

cedentes y se han dejado muchos vacíos en el conocimiento, pero un estudio profundo de esta temática está fuera de los objetivos del texto.

Tipos de sistemas operativos

El desarrollo del *hardware* y las necesidades de los usuarios han hecho surgir varios tipos de SO que se detallan a continuación. Las clasificaciones se ajustan a diferentes aspectos:

- De acuerdo a la cantidad de usuarios que pueden usarlo a la vez.
  - Monousuario (*single user*), solamente un usuario puede conectarse al sistema; ejemplos: Windows 95, Windows 98, Windows Me (Lezcano Brito, 2018).
  - Multiusuario (*multi user*), varios usuarios pueden estar conectados y trabajando con el SO en el mismo momento; ejemplos: Ubuntu, Debian y en general los sistemas tipo Unix.
- Tomando en cuenta la cantidad de procesos que se ejecutan a la vez:
  - Multiprogramados o multitarea (*multiprogramming, multitasking*). En estos SO muchos procesos se pueden ejecutar "a la vez" compartiendo el tiempo de un único procesador (no necesariamente de forma equitativa), para lo que es imprescindible que existan varios procesos cargados en memoria; ejemplos: Windows 10, Linux.

Los SO de tiempo compartido y los de tratamiento por lotes multiprogramados constituyen un caso particular de este tipo de SO. En el primero el tiempo se reparte a intervalos iguales, mientras en el segundo se utilizan los intervalos de E/S de un proceso para asignarle el procesador a otro.

  - Monoprogramados o monotarea (*mono programming*, mono tasking). Solamente permiten que se esté ejecutando un proceso, el cual es dueño de toda la memoria de la computadora; ejemplos: MS-DOS, CP/M (Lezcano Brito, 2018).
- Por la cantidad de procesadores que puede explotar el SO:
  - De multiprocesamiento (*multiprocessing*). El SO es capaz de explotar varios procesadores; ejemplos: Windows (8, 10), Red Hat, Fedora, entre otros. Debe observarse que un SO multiprocesamiento

es también de multitarea ya que puede ejecutar varios procesos "a la vez" en cada procesador.

- De monoprocesamiento (*single processing*), solamente pueden explotar un procesador, aunque la computadora sobre la que estén instalados tenga más de uno; ejemplos: Windows 95, Windows 98, entre otros (Lezcano Brito, 2018).

Por último, existen los SO de tiempo real (*real time*), lo cuales deben dar respuestas rápidas, inmediatas o casi inmediatas; ejemplos: QNX, RTLinux (Lezcano Brito, 2018).

Debe observarse que un SO puede incluirse dentro de varias categorías; por ejemplo, el SO Linux es de multiprocesamiento, multiprogramado y multiusuario. La arquitectura de los procesadores modernos se basa en dos concepciones:

- Procesador simple: Un único procesador está contenido dentro de un *chip*.
- Procesador múltiple (denominado *socket*): Un *chip* contiene varios procesadores (se nombran núcleos o *core*); cada núcleo puede tener múltiples procesadores lógicos.

Los procesadores también pueden tener diferentes modos de ejecución, que establecen diferentes niveles de privilegio con relación a lo que se permite hacer; por ejemplo, es obvio que no puede permitirse que cualquier proceso altere el contador de programa, el cual es el registro del procesador que contiene la dirección de la próxima instrucción a ejecutar, ya que eso ocasionaría errores en el orden de ejecución de las instrucciones causando resultados impredecibles.

El modo de operación menos privilegiado del procesador se conoce como modo usuario, los programas casi siempre ejecutan en este modo; el nivel más privilegiado se nombra modo sistema (también se denomina modo núcleo o *kernel*) en el modo *kernel* el *software* tiene control absoluto sobre los recursos del sistema (Stallings, 2014). Esta concepción permite que el SO pueda protegerse a sí mismo y a los procesos.

Los procesadores contienen diferentes registros destinados a fines específicos, ya se mencionó el contador de programa (CP), pero existen otros, tales como, el registro de instrucciones (RI) que se utiliza para

cargar la instrucción en sí, entre otros.

La planificación y los procesos

Los equipos de cómputo y los SO utilizan interrupciones, para avisar de la ocurrencia de algo; por ejemplo, la terminación de una entrada o de una salida. Las interrupciones se usan para coordinar muchas de las tareas que realiza o monitorea cualquier SO.

Si el SO quiere atender una interrupción que ha recibido, deberá guardar alguna información que le permita atenderla y después regresar a lo que estaba haciendo cuando fue interrumpido en el pasado; por ejemplo, es imprescindible guardar el valor de todos los registros del procesador, en especial el registro contador de programa, las direcciones de memoria del proceso, entre otros. La información guardada se denomina contexto de ejecución del proceso y deberá utilizarse alguna estructura de datos para localizarla.

En los sistemas multiprogramados es necesario planificar el uso del procesador de manera que se reparta de la mejor manera posible y se use de forma eficiente, para hacer esa tarea el sistema operativo dispone de un componente que se denomina el planificador. La tarea de planificar el uso del procesador la realiza un módulo o subsistema del SO que será objeto de estudio del capítulo I.

La memoria y su manejo

El SO y los procesos de usuario necesitan memoria para poder hacer todas sus tareas. Parece ser que este recurso nunca será suficiente y por eso han surgido diferentes técnicas para administrarla, lo cual será el tema del capítulo II. Por ahora es importante señalar que el SO debe garantizar las siguientes funcionalidades:

- Proteger los espacios de memoria, no permitiendo que un proceso use el espacio de otro, salvo cuando el programador desee explicítame compartirla y tenga derecho para hacerlo.
- Manejar la memoria de manera dinámica, asignándola y liberándola de acuerdo a las necesidades de los procesos y de forma transparente para los que la usan.
- Proveer medios de almacenamiento permanente o que mantengan la información por largos períodos de tiempo, para lo cual deberá interactuar muchas veces con el sistema de archivo.

La protección y la seguridad

La protección y la seguridad juegan un rol importante en cualquier sistema de cómputo. Hoy en día se hace más evidente esta necesidad debido a que la mayoría de las computadoras están conectadas a redes (incluida Internet), lo que permite accederlas desde diferentes partes, haciéndolas vulnerables si no se toman las medidas adecuadas.

El concepto de protección tiene que ver con el control de acceso al sistema y limita los tipos de acceso que se permiten sobre los archivos (quiénes acceden y cómo lo hacen), asegura que sólo los procesos que tienen una autorización apropiada, dada por el SO, operen sobre los segmentos de memoria, el procesador y los demás recursos.

La protección es un problema estrictamente interno que responde a la pregunta, ¿cómo se puede controlar el acceso a los datos y programas almacenados en una computadora?, esa protección se garantiza a través de un mecanismo que controla el acceso, de procesos y usuarios, a los recursos, tal mecanismo debe proveer un medio para imponer el control.

La seguridad establece la autenticación de los usuarios para proteger la integridad de la información (datos y código) del sistema y de los recursos físicos. El acceso controlado no permite los usos indebidos, la destrucción o alteración maliciosa de los datos ni las inconsistencias provocadas accidentalmente.

Planificación y manejo de los recursos

Como ya se ha dicho, existen muchos recursos (de *hardware* o de *software*) asociados a una computadora, el SO es el responsable de manejar esos recursos planificando su uso de la manera más apropiada y equitativa posible, para hacerlo se utilizan diferentes políticas, las cuales deben cumplir los requisitos siguientes:

- Ser imparciales: Significa que se dé servicios iguales, o al menos aproximadamente iguales, a peticiones semejantes.
- Poder diferenciar las respuestas: Ser capaz de discriminar entre diferentes tipos de trabajos que tienen distintas necesidades de servicios; por ejemplo, cuando el SO recibe una interrupción de E/S a disco pudiera privilegiarla debido a que, al final, la mayor parte del trabajo lo hará

el controlador del disco y por eso será pequeña la interrupción del proceso que tiene el procesador en ese instante.

- Ser eficiente: Se traduce en maximizar los procesos terminados por unidad de tiempo (throughput); minimizar el tiempo de la primera respuesta de los procesos interactivos y minimizar el tiempo de espera de los procesos por determinado recurso. Es difícil obtener todo esto a la vez y por eso se investiga constantemente en este campo.

Tendencias de diseño de los SO

Núcleo monolítico

Muchos SO de épocas pasadas tenían un núcleo monolítico (figura 3) en el que se agrupaban todas sus funcionalidades. Los SO con este tipo de núcleo son muy difíciles de mantener, por ejemplo, cuando se modifica un módulo para añadir alguna funcionalidad o corregir algún problema de programación, será necesario recompilar el núcleo completo.

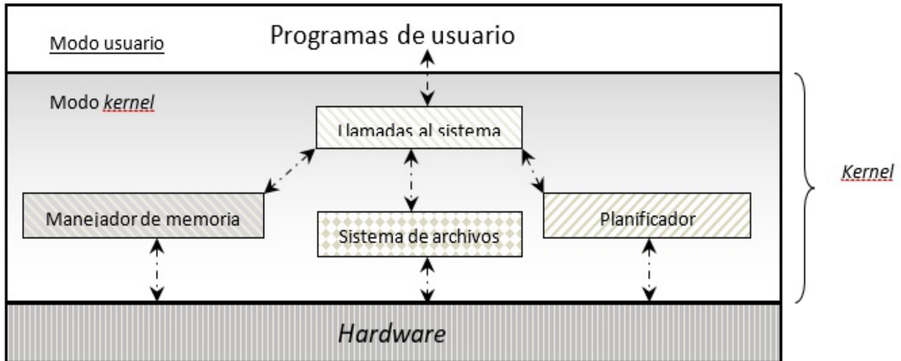


Figura 3. SO (hipotético) con *kernel* monolítico.

Muchos de esos grandes núcleos tienen exceso de funcionalidades que podrían situarse en niveles superiores. Este tipo de sistema se implementa, por lo regular, como un proceso simple que comparte la memoria para todos sus componentes.

En la figura 3 se aprecia que para usar las funciones del núcleo se usan las llamadas al sistema, las cuales pudieran accederse desde una interfaz que se sitúe a nivel del modo usuario.

Micro núcleo (microkernel)

Una idea más avanzada es la arquitectura de micro núcleo (figura 4) en

la cual existe un pequeño núcleo que solo contiene algunas funciones esenciales, como son: la comunicación entre procesos o IPC el espacio de direcciones y algunas funciones básicas de planificación.

Las demás funcionalidades que ofrece el SO se denominan servicios y se implementan en otra capa, nombrada capa de servidores, los servicios se ejecutan en modo usuario de manera que el SO los trata igual que cualquier otra aplicación. De esta forma el núcleo es más pequeño y posee menos complejidad, lo que facilita su actualización cuando es necesario. Por otra parte los servidores se pueden especializar para las exigencias de alguna tarea específica; todo lo cual brinda mucha flexibilidad. Estos aspectos son muy apreciados a la hora de desarrollar sistemas distribuidos.

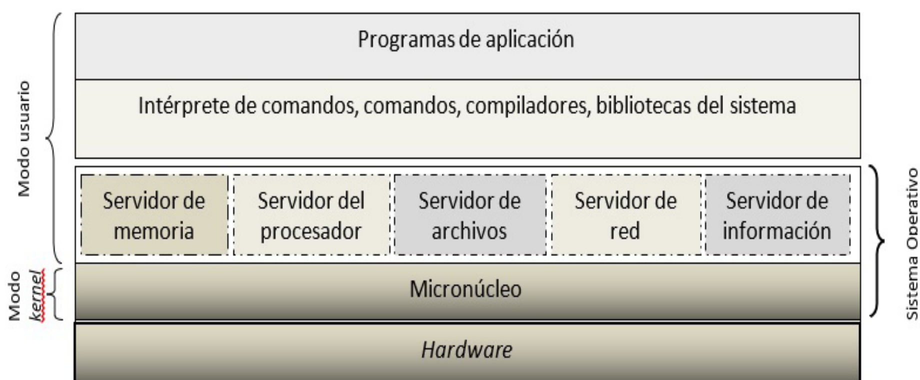


Figura 4. SO (hipotético) con micro núcleo.

La figura 4 muestra un SO hipotético que usa arquitectura de micro núcleo, la capa de servidores contiene los servicios básicos para manejar: la memoria, los procesos, los archivos, la red, así como el sistema estadístico. Esta capa actúa en modo usuario.

Sistemas operativos distribuidos

Un sistema operativo distribuido es una colección de procesadores que no comparten la memoria ni el reloj. Cada procesador tiene su propia memoria local y los procesadores se comunican entre sí a través de líneas de comunicación como puede ser una red de computadoras (Silberschatz, 1996).

El sistema distribuido puede estar constituido por diversos medios de

cómputo, tales como: sistemas de tiempo real, estaciones de trabajo, computadoras centrales (*mainframe*), entre otros., y posee un sistema de archivo distribuido que permite a los usuarios abstraerse de la localización real de sus archivos.

Los usuarios ven un solo espacio de direcciones de memoria principal y un solo espacio de almacenamiento secundario (sin importar su localización real) más algunas facilidades de acceso.

Sistemas operativos con diseño orientado a objetos

La tecnología orientada a objetos ha proliferado y se ha hecho presente en casi todos los diseños actuales, el diseño de los SO no podía escapar a esta tendencia y por eso hoy en día existen SO que utilizan las ideas de este paradigma de programación.

Los sistemas operativos actuales

Los SO más populares al momento de escribir este libro (2019) son los diversos Windows de Microsoft y los tipos Unix, ellos son capaces de explotar pastillas (*chips*) que tienen billones de transistores y trabajan con unidades de procesamiento central (CPU) de 32 y 64 bits, además de otras unidades de procesamiento como la GPU (Graphical Processing Unit) y la DSP (*Digital Signal Processors*).

Estos SO tienen que ser capaces de proveer apoyo para usar una gama amplia de unidades de E/S, entre las que se incluyen: cámaras, monitores sensitivos al contacto, micrófonos, sensores especializados, entre otros. Los SO también tienen que lidiar ahora con una amplia variedad de datos, tales como: fotos, mapas de distintos tipos, entre otros.

Todo lo anterior y, muchas otras cosas que la ciencia y el diario quehacer necesitan manejar, hace que el campo de acción de las computadoras y por ende de los SO se haya ampliado de una manera increíble y por eso es necesario que se usen múltiples CPU, GPU y DSP para realizar los trabajos que hoy se le encomiendan a diversos equipos de cómputo.

Los sistemas operativos Windows y tipo Unix

Una vez analizadas algunas generalidades que atañen a cualquier SO, es el momento adecuado para enfatizar en algunas particularidades de dos tipos de SO que son muy populares hoy en día. El pequeño rela-

to comienza por el más antiguo de los dos.

Sistemas operativos tipo Unix. Una vista general

La primera versión de UNIX data del año 1969 y se hizo en los laboratorios Bell (una parte de AT&T) con el objetivo de explotar una mini computadora PDP-7, posteriormente se extendió hasta la PDP-11 (figura 5).

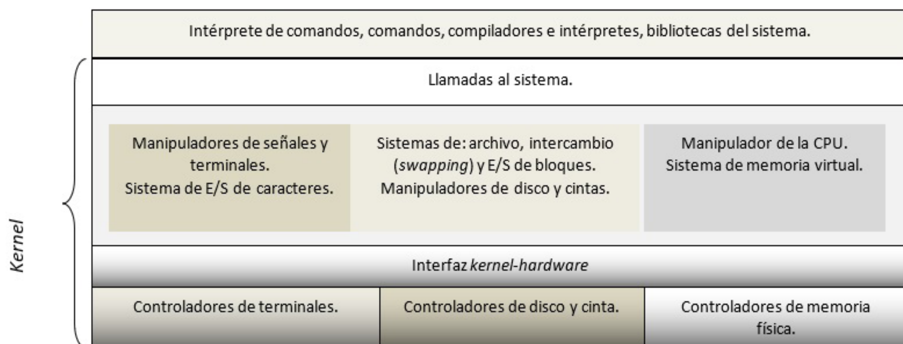


Figura 5. Arquitectura monolítica de UNIX.

Quizás para al lector no signifique mucho que el SO se implementara para dos máquinas diferentes pero ese hecho fue algo muy notable en aquella época porque ya se tenía un SO capaz de explotar distintos procesadores. Sus principales autores fueron Ken Thompson y Dennis Ritchie quienes fueron Premios Turing en 1983.

En la figura 5 se muestra la arquitectura original del SO UNIX. La primera versión del SO se escribió en lenguaje ensamblador, pero ya para la tercera versión la mayoría del código estaba escrito en el lenguaje C, que fue (Lezcano Brito, 2018) especialmente desarrollado con ese fin, de ahí que las historias del C y del UNIX estén unidas muy estrechamente. El hecho de escribir, por primera vez, un SO en un lenguaje de alto nivel fue uno de los logros notables de este diseño, hoy en día casi todas las versiones de UNIX (y de muchos SO) están escritas en C.

El SO era simple con relación a otros SO que influyeron en su diseño: Multics (*xed Information and Computing Services*, fue uno de los primeros sistemas de tiempo compartido) y CTSS. UNIX emergía dividido en módulos pequeños que tenían la responsabilidad, según palabras de sus

autores, de *"…hacer cada uno de ellos una única función, pero hacerla muy bien…"*. La primera versión de UNIX era para un solo usuario (mono usuario).

En 1976 se usó la primera versión (la 6) de este SO fuera de los laboratorios Bell, a la que siguió la versión 7 en 1978, esta última ha sido la antecesora de muchos de los SO UNIX y tipo Unix modernos.

En esta breve historia debe mencionarse el SO UNIX BSD (*Berkeley Software Distributio*n), que fue el primero de este tipo de SO que se hizo fuera de los laboratorios Bell (se ejecutaba sobre computadoras PDP y VAX). Como una muestra del desarrollo en ciclo, Bell tomó algunas ideas del BSD, las refinó y las incorporó al sistema UNIX System III (1982), hasta obtener el UNIX System V.

Los SO tipo Unix han evolucionado constantemente y por eso han proliferado múltiples implementaciones que se pueden ejecutar sobre diversas plataformas, ellas han incorporado constantemente, las innovaciones que van surgiendo en las investigaciones sobre SO que se apoyan en el desarrollo del hardware. Las arquitecturas de estas nuevas implementaciones se caracterizan por tener un pequeño núcleo que proporciona funciones y servicios a otros procesos del SO.

Algunos ejemplos de SO tipo Unix actuales son:

- System V Release 4 (SVR4). Desarrollado por AT&T y Sun Microsystems, combinando facilidades de SVR3, 4.3BSD, Microsoft Xenix System V (Stallings, 2014) y SunOS. El sistema incluye: soporte para procesamiento en tiempo real, asignación dinámica de estructuras de datos, manejo de memoria virtual, sistema de archivo virtual y su núcleo puede desalojarse.
- BSD: es de destacar la incidencia que ha tenido este sistema en las investigaciones acerca de SO y su versión 4.xBSD se usa en ambientes académicos y productivos. Las versiones FreeBSD, basada en servidores Internet y corta fuegos, se usa en muchos sistemas empotrados (*embedded*) y el Mac OS X (*Macintosh Operating System*), se basa en FreeBSD 5.0 con el micro núcleo Mach 3.0.
- Solaris 10. Está basado en SVR4 y por eso proporciona todas sus facilidades, pero adiciona otras características más avanzadas, como son: un núcleo multihilo que se puede desalojar, soporte para SMP (*Symmetric*

Multi Processing o Multiprocesamiento simétrico, que es una arquitectura de computadora que permite compartir una misma memoria entre varios procesadores.) e interfaz orientada a objetos para el sistema de archivo.

- Linux. La versión inicial la hizo Linus Torvalds, como una variante de UNIX para la IBM PC (computadora con arquitectura Intel 80386). Su autor la liberó a Internet (1991) y pidió cooperación para seguir su desarrollo siguiendo las ideas del *software* libre. Hoy en día este tipo de sistema ha logrado una gran popularidad y se ha extendido a muchas plataformas con diferentes implementaciones.

Sistemas operativos Windows. Una vista general

Bajo el nombre *Windows* se agrupa un conjunto de SO multitareas que ofrece una plataforma cómoda y amigable para usuarios y desarrolladores. Las primeras versiones se hicieron a partir del SO MS-DOS (*Microsoft Disk Operating System*) que era un sistema mono usuario y mono programado con interfaz de línea de comandos hecho en lenguaje ensamblador.

Originalmente Windows se programó para una arquitectura Intel i860, pero hoy en día ya está implementado para otras plataformas de *hardware*; por ejemplo, para los procesadores: *Digital Alpha, PowerPC, MIPS, Intel IA64 (Itanium)* y las versiones de 64 bits del x86 basadas en AMD.

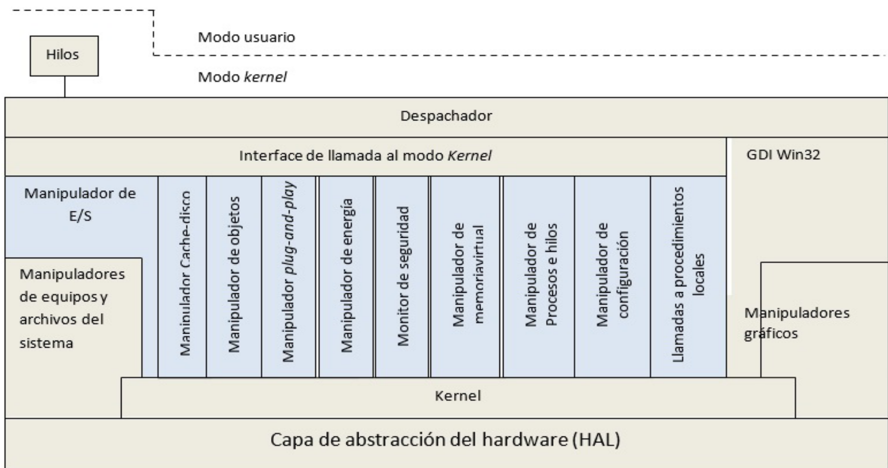


Figura 6. Arquitectura del núcleo de Windows.

La figura 6 muestra la arquitectura del núcleo de una de las versiones de *Windows*. La primera capa de la figura es la de abstracción del *hardware*. Su responsabilidad es mapear comandos genéricos de *hardware* en respuesta a una petición específica, lo que permite aislar al SO de las diferencias que existen entre distintas plataformas, logrando que luzcan igual para el *kernel* y la parte ejecutiva.

El *kernel,* controla la ejecución de los procesadores, maneja la planificación de los hilos, el intercambio de procesos, manipula excepciones e interrupciones y sincroniza el multiprocesamiento. Este parte no se ejecuta como hilo.

La parte señalada en azul se denomina Ejecutiva. A continuación, se detallan, brevemente sus funcionalidades:

- Manipulador de E/S: brinda un medio para que las aplicaciones accedan a los equipos de E/S, enviando las peticiones a los manipuladores de equipos (device drivers) adecuados.
- Manipulador caché-disco: hace eficiente las E/S a archivos ubicando los datos más recientemente referenciados en memoria principal y mantiene las escrituras destinadas a disco en memoria, por cortos períodos de tiempo antes de enviarla al disco.
- Manipulador de objetos: crea, manipula y borra los objetos ejecutivos Windows, los cuales se usan para representar los recursos, por ejemplo: procesos, hilos, objetos de sincronización, entre otros.
- Manipulador *plug-and-play*: determina y carga el manipulador que se necesita para operar un equipo que se ha agregado al sistema de cómputo.
- Manipulador de energía: coordina el manejo de la energía de los equipos, lo que permite configurarlos para reducir el consumo.
- Monitor de seguridad: hace cumplir reglas de validación de acceso y genera auditorías, lo cual se facilita por el modelo orientado a objetos del SO.
- Manipulador de memoria virtual: maneja el espacio de direcciones virtuales y físicas, así como la paginación. Controla el *hardware* de memoria y las estructuras de datos que mapean direcciones virtuales en direcciones físicas.
- Manipulador de procesos e hilos: crea, manipula y borra procesos e hilos.

- Manipulador de configuración: implementa y maneja el registro del sistema.
- Llamadas a procedimientos locales: implementa un mecanismo eficiente para la comunicación, los servicios de procesos locales y los subsistemas.

La capa de manipuladores de equipos y archivos del sistema, es una biblioteca dinámica que extiende la funcionalidad de la parte ejecutiva e incluye: manipuladores de equipos que traducen las llamadas a funciones de E/S de los usuarios a los equipos específicos y a los componentes de *software* para implementar el sistema de archivo, los protocolos de red y muchas otras extensiones que se ejecutan en modo *kernel*.

Los sistemas gráficos y GDI (componente o subsistemas de la interfaz de usuario de Windows) Win32 implementan las funciones GUI (Graphical User Interface), como son: el trabajo con ventanas, el control de la interface de usuario y el dibujo.

Organización de los contenidos

Hasta este punto, se ha pretendido dar una visión general de los SO para entender cada uno de los capítulos que siguen. Muchos de los aspectos que se presentaron en esta breve introducción se retomarán más adelante.

El capítulo I, "Procesos e hilos", trata acerca de las distintas formas que existen para planificar el uso del procesador central. Explica los diferentes tipos de planificadores y los algoritmos que se usan para distribuir el tiempo de procesamiento (Lezcano Brito, 2018). También se analizan los problemas de sincronización entre procesos concurrentes los cuales ejecutan múltiples tareas interactuando entre sí.

En el capítulo II, "Administración de la memoria", se analizan las formas más comunes de administración de la memoria, el paginado y la segmentación, los algoritmos asociados a esta actividad y las estructuras de datos que se utilizan para controlar el uso de la memoria.

El capítulo III, "Equipos de almacenamiento masivo" brinda detalles acerca de los equipos de almacenamiento masivo y da pie a la lectura del capítulo IV.

En el capítulo IV, "Sistema de archivos", se estudia el concepto abstrac-

to de archivo y las formas organizativas de este subsistema: contigua, enlazada e indexada.

Todos los capítulos finalizan con un pequeño resumen, un conjunto de ejercicios y relacionan la bibliografía básica que han tomado en cuenta los autores para escribir este texto, aunque es casi seguro que muchas explicaciones están influidas, quizás de manera inconsciente, por otra literatura leída durante sus largas vidas como profesionales de la computación.

El libro finaliza con tres anexos que ayudan a completar la información. El anexo 1 presenta algunos comandos de los SO tipo Unix, el anexo 2 se dedica a comandos de los SO Windows y el anexo 3 versa acerca de las unidades que se utilizan para medir información.

Capítulo I. Procesos e hilos

1.1. Los procesos en un equipo de cómputo

Todos los procesos que actúan en un equipo de cómputo necesitan usar el procesador central o CPU y por eso el SO debe establecer mecanismos que permitan planificar este recurso de *hardware* de manera adecuada, para lo cual es necesario establecer diversas reglas y procedimientos. El objetivo principal de este capítulo se enfoca en esos detalles y en los conflictos que surgen al compartir el procesador.

Los SO multitarea se caracterizan por compartir un único procesador central entre varios procesos. Es importante no asociar la palabra compartir, en este contexto, con homogeneidad, excepto en los casos que se diga explícitamente.

Los SO multiprocesamiento permiten explotar equipos de cómputos con varios procesadores, cada uno de ellos puede planificarse para que varios procesos lo usen, o sea en estos sistemas la multitarea se da a nivel de cada procesador.

Al compartir los recursos de un sistema de cómputo debe garantizarse la protección de los datos, del código y de cualquier otro medio; en ese sentido puede pensarse que el SO es una capa de abstracción que provee un medio para usar esos recursos.

En el procesador central se efectúa la mayoría de las operaciones que realizan las computadoras, aunque también existen otros procesadores con fines más específicos; por ejemplo, la GPU y la DSP, la primera encargada de procesar gráficos y la segunda para trabajar con señales streaming, que es una transmisión de lectura continua que permite descargar un contenido y verlo a la vez (en paralelo), tales como el video y el audio.

Aparte de su propósito, La GPU también permite realizar cálculos eficientes sobre arreglos de datos, usando SIMD (*Single-Instruction Multiple Data*, donde todos los procesadores de un sistema ejecutan el mismo torrente de instrucciones sobre diferentes piezas de datos), por ese motivo hoy en día se usa también para procesamiento numérico y no solo renderizar gráficos que significa generar imágenes o videos partiendo

de un modelo en 3D, por otra parte la DSP, que se usa embebida en equipos de E/S, se está convirtiendo también en un equipo de cálculo.

Además de la GPU y la DSP también existen otras unidades especializadas para trabajar con la codificación/decodificación (codecs) de videos, para encriptar y desencriptar, entre otros.

Los sistemas conocidos como sistemas sobre un *chip* SoC se han hecho populares hoy en día; ellos se caracterizan porque un solo *chip* contiene varios componentes como pueden ser: un microprocesador clásico, memoria caché y principal, DSP, GPU, equipos de E/S (ej. radio); todo lo cual ayuda a satisfacer las necesidades de los equipos manuales, un ejemplo de este tipo de sistema en el teléfono celular.

La ejecución de programas y las interrupciones

- La ejecución de un programa puede verse como un ciclo de búsqueda y ejecución de instrucciones como el que se muestra en el algoritmo de la figura. La ejecución de cada instrucción puede incluir varias operaciones, a ese conjunto de operaciones se le denomina ciclo de instrucción.

```
Repetir
Buscar próxima instrucción <I> en memoria
    Si existe <I>
      Cargar <I> en Registro de Instrucciones (RI)
      Interpretar la acción contenida en <RI>
      Realizar la acción interpretada
Hasta que no exista <I>
```

Figura 7. Algoritmo de ejecución de un programa.

Cada vez que se termina de buscar una instrucción, el procesador incrementa en uno el registro contador de programa (al menos que se le diga lo contrario). Las instrucciones están divididas en partes (al menos dos), en la primera aparece el código de la operación a realizar y en la segunda su dirección.

Las acciones referidas en el algoritmo anterior pueden categorizarse de la manera siguiente (en forma general):

- Acción procesador-memoria. Transfiere datos desde el procesador hacia la memoria o viceversa.

- Acción procesador-E/S. Transfiere datos entre un periférico de E/S y el procesador.
- Acción de procesamiento. El procesador realiza alguna operación aritmética o lógica sobre los datos.
- Acción de control. Puede afectar la secuencia de ejecución; por ejemplo, si es una instrucción de salto.

El seudocódigo de la figura 7 funciona sin problemas, pero si se efectúa un ciclo de E/S el procesador tendrá que esperar constantemente por el equipo involucrado en la operación. Para resolver ese problema se usan las interrupciones de la manera siguiente:

- El procesador envía una solicitud al equipo de E/S y continúa haciendo sus operaciones.
- El equipo prepara el conjunto de datos y lo pone en alguna parte, cuando termina de depositar los datos le avisa al procesador (a través de una interrupción).
- El procesador recibe la interrupción y la atiende tomando los datos preparados por el equipo de E/S.

Esta forma de trabajo permite que el procesador y el equipo de E/S trabajen concurrentemente. Las interrupciones pueden ser de diversos tipos, por ejemplo: disco lleno, datos listos, se terminó el papel, fin de la impresión, entre otros.

Desde el punto de vista de un programa de usuario, las interrupciones suspenden temporalmente la ejecución del proceso y cuando esa interrupción se trata (se hace lo que ella necesita), el proceso continúa su secuencia de ejecución. Se puede observar que el proceso no contiene ningún código específico para hacer la tarea encomendada (en este caso la E/S) debido a que esa es una tarea del SO.

Las interrupciones pueden estar habilitadas o no, cuando están deshabilitadas no es posible atenderlas. Para procesarlas se pueden utilizar dos alternativas:

- Antes de comenzar a tratar una interrupción se deshabilitan las demás. De esta forma el procesador solo atenderá la que está habilitada, si durante ese tiempo arriba otra se deja pendiente.

Esta estrategia no toma en cuenta que existen prioridades críticas, dependientes del tiempo, que deberían atenderse lo antes posible debido

a que ese tiempo de desatención puede comprometer los resultados bajo ciertas circunstancias.

- La segunda alternativa es asociar prioridades a las interrupciones, de manera que si se está atendiendo una interrupción y llega otra con mayor prioridad se deje pendiente la actual para atender la que arribó.

1.2. Paralelismo

Esta sección ofrece una vista general de algunas configuraciones de *hardware* que apoyan el procesamiento paralelo, se hace con el único propósito de facilitar las explicaciones relacionadas con el tema, pero el estudio avanzado de tales tópicos no se incluye entre los objetivos del libro.

Resulta usual ver las computadoras como máquinas secuenciales en las cuales los procesos siguen una secuencia de instrucciones establecidas por un algoritmo dado. Sin embargo, esa visión no es correcta, sobre todo porque el desarrollo del *hardware* y la reducción de su costo han facilitado el diseño de sistemas de cómputo que permiten hacer tareas en paralelo. En esa área pueden citarse: el multiprocesamiento simétrico (SMP), las computadoras con más de un procesador (multiprocesadores) y las que tienen varios núcleos dentro de un *chip* (multinúcleo).

Multiprocesamiento simétrico (SMP)

Una computadora con arquitectura SMP se caracteriza por:

- Poseer dos a más procesadores con capacidades similares.
- Los procesadores comparten la memoria principal; las facilidades de E/S están interconectadas de manera que el tiempo de acceso a memoria es aproximadamente el mismo para cada uno de los procesadores.
- Todos los procesadores comparten el acceso a los equipos de E/S, a través de un mismo canal o de canales diferentes.
- Todos los procesadores resuelven las mismas funciones.
- El sistema está controlado por un SO integrado que provee la interacción entre los procesadores, sus procesos y sus datos (figura 8).

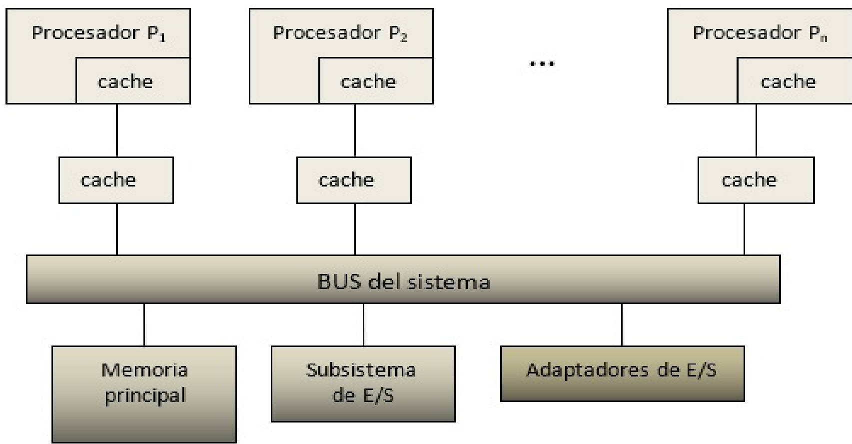


Figura 8. Organización de un sistema SMP.

Como se aprecia el multiprocesamiento simétrico es una capacidad del *hardware* y los SO que se diseñan para explotarla también reciben ese nombre. El *hardware* de las computadoras con multiprocesamiento simétrico permite alcanzar los siguientes objetivos:

- Ser eficiente: el hecho de que existan varios procesadores permite que las tareas independientes puedan ejecutarse sobre distintos procesadores, en este caso se habla de multiprocesamiento y no de multitarea.
- Confiabilidad: cuando un procesador falla, otro puede asumir esas funciones, aunque en ese caso se reducirá la eficiencia del sistema.
- Crecimiento incremental: los usuarios pueden agregar más procesadores a la arquitectura.
- Escalables: los fabricantes pueden ofrecer distintos rangos de productos en dependencia de las necesidades de los usuarios y el costo.

Los SO SMP tienen aplicaciones en tareas que se caracterizan por estar formadas por sub-tareas independientes no serializadas; por ejemplo, los servidores de bases de datos que se mantienen a la escucha de distintos procesos clientes.

En este tipo de SO los procesos se planifican para ejecutarse sobre varios procesadores que forman parte de la arquitectura del *hardware* subyacente.

Multinúcleo

Una computadora multinúcleo (*multicore*) posee una pieza simple de silicón con múltiples procesadores en su interior. Tiene núcleos complejos, basados en arquitecturas con procesadores tradicionales; normalmente tienen memoria compartida, con múltiples capas de cachés, y se utilizan como procesadores independientes (Dawwd, 2019).

El propósito es explotar el paralelismo sin necesidad de fabricar procesadores más complejos, se puede citar como ejemplo, el procesador Intel Core i7 (figura 9), que contiene cuatro procesadores x86 cada uno de ellos con una caché dedicada y una compartida entre todos los núcleos.

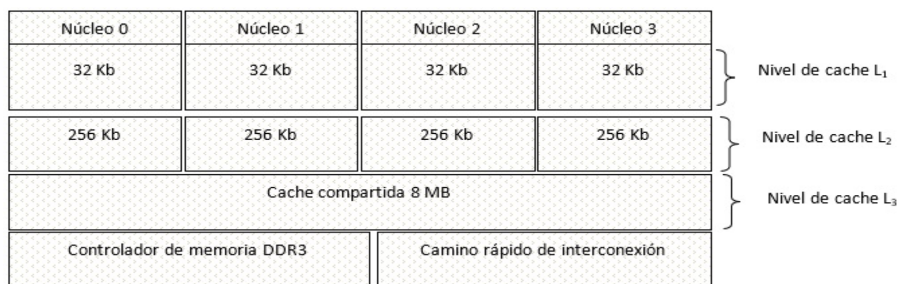


Figura 9. Diagrama de bloques del procesador i7.

Este procesador proporciona dos maneras de comunicación externa:

- La primera a través del controlador de memoria DDR3.
- La segunda a través del camino rápido de interconexión, denominado QPI (Quick Path Interconnect) que ofrece una comunicación de alta velocidad.

1.3. Planificación de procesos

Los procesos son ente activos y esa realidad queda implícita en la definición que se le ha dado hasta el momento -"un proceso es un programa en ejecución"-; es decir es una entidad que realiza diversas acciones para las cuales necesita varios recursos (Lezcano Brito, 2018), esta última afirmación refleja el hecho de que los procesos son propietarios de algunos recursos, por ejemplo: un espacio de direcciones virtuales que le permite almacenar su código y sus datos, determinados archivos abiertos, algún espacio de memoria física, entre otros.

En los sistemas multiprogramados varios procesos compiten por el procesador, es por eso que es necesario decidir a qué proceso se le debe

entregar el procesador (planificar su uso), después de tomada esa decisión deberán hacerse algunas acciones para entregar el procesador al proceso escogido (despacharlo).

Tomando en cuenta todos los detalles apuntados anteriormente se puede redefinir el concepto de proceso de la forma siguiente: un proceso es un programa en ejecución y una unidad de planificación y despacho (Lezcano Brito, 2018) que posee los recursos necesarios para realizar su trabajo.

Debe insistirse acerca de la diferencia que existe entre programa (ente pasivo) y proceso (ente activo). Todo proceso se crea a partir de un programa ejecutable que se convierte en proceso cuando el SO lo carga en memoria (parcial o totalmente) y construye (Lezcano Brito, 2018) una estructura de datos para controlarlo, denominada bloque de control de proceso (figura 10) o simplemente PCB.

PID
Estado
Contador de programa
Punteros a direcciones de memoria
Valores de los Registros
Puntero a la tabla de archivos abiertos
...

Figura 10. PCB típico.

Existe un PCB por cada proceso, el cual puede contener diversos campos en dependencia de cada SO, pero resulta imprescindible que contenga:

- Un identificador que distingue al proceso de los demás. Usualmente es un número entero conocido por sus siglas en inglés PID.
- Su estado. Define en una palabra que está haciendo el proceso en ese instante (ejecutando, detenido, entre otros) (Lezcano Brito, 2017).

- El contador de programa. Contiene la dirección de la próxima instrucción que ejecutará el proceso y se toma del valor del registro contador de programa en el momento en que el proceso se interrumpe.
- Los valores de los demás registros del procesador. Se refiere a los valores de esos registros cuando el proceso fue interrumpido
- Sus direcciones de memoria. El lugar donde está el código y sus datos.
- Los archivos abiertos. Se utiliza una tabla para controlar los archivos abiertos.
- La pila del proceso.

Para intercambiar el uso de la CPU entre distintos procesos en un sistema multiprogramado se sigue la rutina siguiente:

- Cada vez que se interrumpe un proceso A (no importa por qué) se guarda toda su información en su PCB (se le denominará PCBA).
- El SO escoge el proceso que ejecutará en lo adelante, sea el proceso B.
- El SO toma los valores del PCBB y carga los registros del procesador con esos valores.
- Después de lo cual le asigna el procesador al proceso B que comenzará a ejecutar.

Debe observarse que la rutina anterior garantiza que un proceso interrumpido pueda continuar por el mismo lugar cuando le asignen de nuevo el procesador.

El PCB puede considerarse la CPU virtual de cada proceso en el sentido que el PCB de un proceso i, PCBi, es una imagen del estado de la CPU para ese proceso. La idea básica para detener procesos e intercambiar las tareas, de modo que teniendo una sola CPU parezca que se ejecutan varias tareas a la vez, se basa en dos aspectos (Lezcano Brito, 2018):

1. Cada vez que a un proceso se le retire la CPU se guardan los valores de los registros del procesador en el PCB del proceso (la CPU virtual del proceso).

2. Cada vez que se le asigne el procesador a un proceso, se toman los valores de su PCB (su CPU virtual) y se copian en los registros correspondientes del procesador de la computadora, con lo cual la CPU

queda tal como estaba cuando el proceso que se va ejecutar fue interrumpido en el pasado.

Intervalos de vida y estados de un proceso

La vida de los procesos se caracteriza por transcurrir en dos intervalos de acciones nombradas ráfagas de CPU y ráfagas (intervalos o etapas) de entrada/salida. La primera se refiere al tiempo en que el proceso es dueño del procesador (Lezcano Brito, 2018), mientras la segunda es cuando el proceso está esperando por la transferencia de datos (de entrada o de salida).

Durante su existencia, los procesos pasan por diferentes estados que se asocian con la actividad que están realizando, algunos de ellos son:

- Nuevo: el proceso acaba de crearse (se le construyó su PCB) pero aún no se ha admitido en el sistema para competir por los recursos.
- Ejecutando: el proceso tiene asignado el procesador y por eso está ejecutando sus instrucciones.
- Listo: en este estado el proceso tiene todo listo para poder ejecutar, pero deberá esperar porque se le asigne el procesador y por eso su PCB permanece en la cola de listos.
- Bloqueado o esperando: el proceso debe esperar por algún evento; por ejemplo, la terminación de una entrada/salida y por eso su PCB está en una cola asociada al equipo de E/S o está realizando la E/S en ese momento.
- Terminado: el proceso ya terminó todas sus acciones y queda en este estado hasta dejar de existir. El SO necesita un tiempo para liberar algunos recursos, por ejemplo, el PCB (Lezcano Brito, 2018).

La figura 11 muestra la relación entre estos estados, la cola de listos, la CPU y los periféricos de E/S.

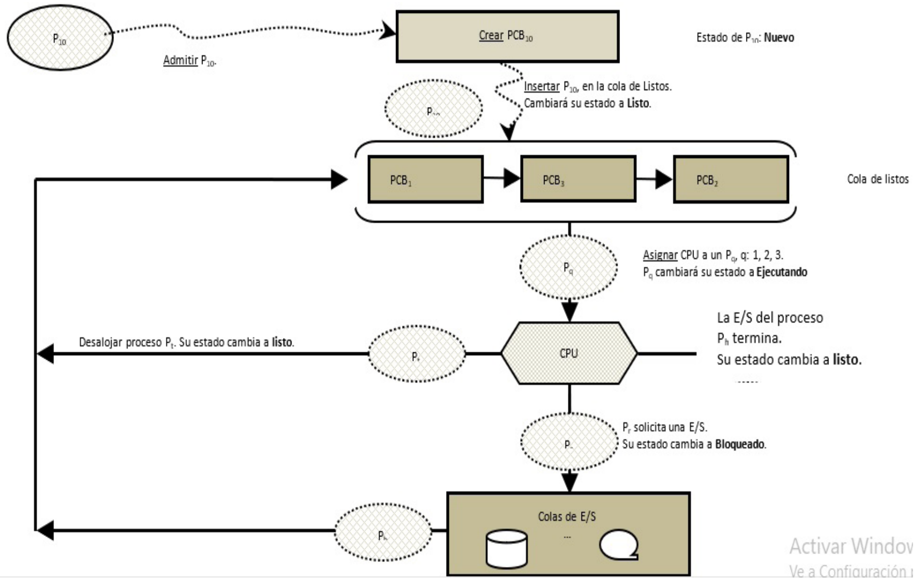


Figura 11. Cola de listos, la CPU y los periféricos de E/S.

Todos los SO multitarea poseen algún planificador que se encarga de decidir cuál será el próximo proceso al que se le asignará la CPU, permitiendo que se comparta el o los procesadores del sistema de cómputo entre un conjunto de procesos. En sistemas con varios procesadores (multiprocesamiento) la multitarea se da a nivel de cada procesador, o sea asociado a cada procesador Pri existe un conjunto de procesos Pj que compite por el procesador.

Durante su vida, los procesos migran entre diferentes colas de planificación, cada una de ellas está controlada por un tipo de planificador dado. La figura 12 muestra la relación entre los tres planificadores: de período largo, de período medio y de período corto.

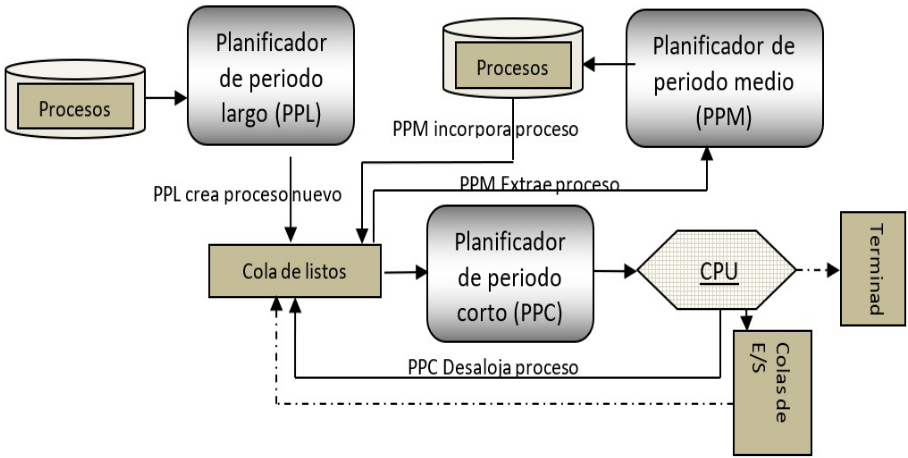


Figura 12. Relación entre los tres planificadores.

En los SO de tratamiento por lotes los procesos nuevos se almacenan en algún dispositivo externo, usualmente un disco. El planificador de periodo largo es el responsable de atender esa cola, cargar los procesos en la memoria e incorporarlos a la cola de listos. Este planificador no está presente en algunos SO; por ejemplo, en UNIX o en los SO de Microsoft.

El planificador de periodo corto selecciona, de la cola de listos, el próximo proceso que usará el procesador.

La frecuencia de ejecución de estos dos planificadores establece una clara distinción entre ellos, mientras el de periodo corto se ejecuta con bastante periodicidad (rango de milisegundos), el de periodo largo lo hace a intervalos más prolongados y tiene la responsabilidad adicional de controlar el grado de multiprogramación (cantidad de procesos en memoria); logrando una buena mezcla de trabajo o sea tener un balance adecuado entre los procesos que usan mucho la CPU se dice que están acotados a CPU y los que realizan muchas E/S se dice que están acotados a E/S.

En algunos SO puede existir un tercer planificador, nombrado de periodo medio, con el propósito específico de mantener el grado de multiprogramación, retirando o adicionando procesos de la memoria.

El planificador de periodo corto y el despachador

En la figura 5 se aprecia que los procesos pueden pasar del estado listo al de ejecutando. Para que esa transición ocurra el planificador de periodo corto, también conocido como planificador de la CPU, deberá seleccionar uno de los procesos candidatos que están en la cola de listos.

Una vez que el planificador de periodo corto ha hecho su elección (basado en alguna estrategia), se la comunica al despachador que será el responsable del cambio de contexto.

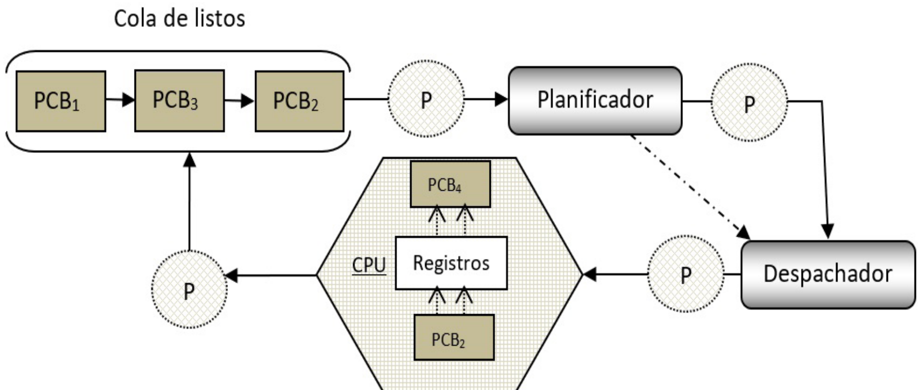


Figura 13. Interacción planificadora de la CPU-despachador.

El cambio de contexto es la tarea que lleva a cabo el despachador para garantizar que el proceso que tenía la CPU, y se le retira en favor de otro, pueda ejecutar en un futuro por donde mismo iba, y también que el proceso que ocupará la CPU a partir de ese momento pueda continuar por el mismo lugar que estaba cuando fue detenido en el pasado, por supuesto que cuando se ejecuta por primera vez no hay tal pasado. Obsérvese que, si el proceso que tenía la CPU ya terminó todo su trabajo, solo habrá que cargar un proceso nuevo.

La figura 13 muestra la idea en forma esquematizada. En este caso hay tres procesos en la cola de listos: P1, P3 y P2, ellos tienen todo lo necesario para ejecutar (están en el estado listo) pero aún no se le ha asignado la CPU. Por otra parte, el proceso P4 es dueño de la CPU (está en estado ejecutando).

La figura muestra el instante en que el planificador de periodo corto decide que P4 ha ejecutado por tiempo suficiente y que le dará el turno al proceso P2. Esa decisión se la comunica al despachador que se encargará del cambio de contexto, para lo cual seguirá la rutina siguiente:

1. Toma los valores de la CPU y los guarda en el PCB del proceso saliente (P4). Le cambia el estado a P4 de ejecutando a listo.
2. Inserta el proceso P4 en la cola de listos.
3. Toma los valores del PCB del proceso P2 y carga la CPU con esos valores. Le cambia el estado a P2 de listo a ejecutando.
4. Le da el control de la CPU a P2.

Los procesos, tal y como se ha visto hasta el momento se conocen también por el nombre de procesos pesados (Lezcano Brito, 2017), el adjetivo pesado hace referencia a la complejidad que trae consigo la creación de un proceso de este tipo. Los procesos pesados siguen una única secuencia de control para ejecutarse y no comparten nada con ningún otro proceso, de manera que la única vía para comunicarse entre ellos es a través de mensajes. El contexto del proceso pesado incluye, entre otras cosas, una pila (stack), referencias a los archivos abiertos, su código y sus datos.

Existe otro tipo de proceso, denominado proceso ligero o hilo, que posee los mismos elementos que los procesos pesados, pero son hijos de algún proceso y comparten algunas cosas con su padre y hermanos.

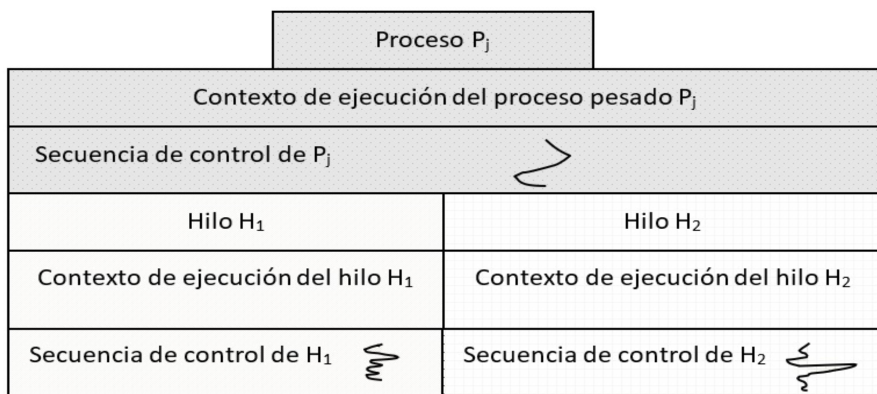


Figura 14. Un proceso pesado con dos hilos.

La figura 14 muestra la idea esquematizada de un proceso pesado (Pj) con dos hilos (H1 y H2). Debe observarse que tanto los procesos pesados como los hilos tienen un contexto de ejecución y una secuencia de control propios.

Los hilos de un proceso pesado comparten con sus hermanos el espacio de memoria del padre y sus archivos abiertos, pero cada hilo tiene su propio espacio de memoria y puede abrir nuevos archivos. La memoria común compartida con el padre se puede usar como vía de comunicación entre ellos.

La creación de un hilo es más rápida que la de un proceso pesado de ahí el adjetivo de ligero.

Algoritmos de planificación

Existen diversos algoritmos para planificar el uso del procesador central o CPU, en general pueden seguir dos políticas con relación a la forma en que se asigna el procesador:

- Sin desalojo: en estos algoritmos cuando un proceso está en el estado ejecutando solo se le retira la CPU (Lezcano Brito, 2018) si solicita alguna E/S, de manera que el planificador de periodo corto solo entra en acción en ese caso o cuando el proceso termina. Los SO Windows 3.x usaban esta forma de planificar.
- Con desalojo: en estos algoritmos se le puede retirar el procesador a los procesos, aunque no hayan finalizado su ciclo de uso de la CPU. De tal forma, a los dos momentos anteriores para planificar un nuevo proceso (terminó, tuvo que hacer una entrada/salida), se le agrega un tercero que es cuando se le quita el procesador al proceso que lo tiene (Lezcano Brito, 2018).

Los motivos para quitarle el procesador a un proceso pueden ser varios, por ejemplo: se le había asignado por un tiempo que llegó a su fin, arribó un proceso de mayor prioridad, entre otros (Lezcano Brito, 2018).

Algoritmo FCFS - El primero en llegar es el primero en ser servido

Este algoritmo le asigna la CPU al primer proceso que haga la solicitud. Es el más simple de todos, pero su rendimiento puede ser bajo. Para implementarlo, se puede usar una cola FIFO, es decir, una cola donde los procesos que arriban al estado de listo se insertan al final y cuando es

necesario escoger uno nuevo para que use la CPU, se extrae el primero de la cola (Lezcano Brito, 2018).

Este algoritmo es, típicamente, sin desalojo dado que no existen criterios para retirarle la CPU al proceso que esté ejecutando. En este aspecto, es importante recordar que los procesos no están siempre haciendo trabajos en la CPU y que su vida transcurre en ráfagas de uso de la CPU y de E/S.

Cuando se usa el algoritmo FCFS y un proceso acotado a CPU se apropia de este recurso, la cola de listos aumenta de tamaño considerablemente, debido a que no hay desalojo y solo se libera la CPU cuando el proceso haga alguna E/S. Este problema provoca esperas prolongadas y se conoce como efecto de convoy.

Algoritmo de Prioridades

Para implementar este algoritmo se asocian prioridades a los procesos y se puede implementar con desalojo o sin desalojo. Si el algoritmo usa el desalojo, el planificador de periodo corto deberá estar atento para verificar las prioridades de los procesos que arriban a la cola de listos, en el caso que llegue uno con mayor prioridad que el que está ejecutando, le retira la CPU para asignársela al proceso que ha llegado.

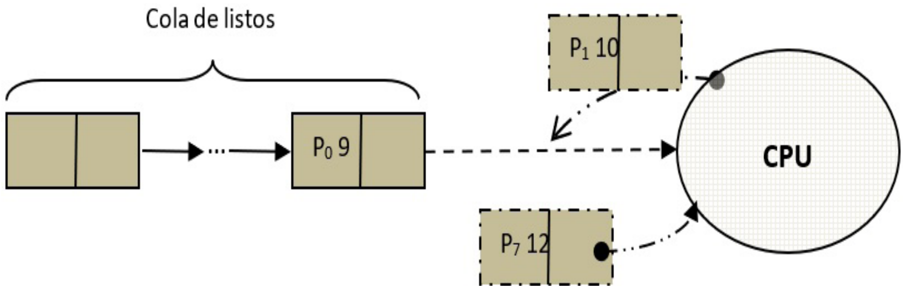


Figura 15. Algoritmo de prioridades (cola ordenada).

La figura 15 refleja el instante de la llegada del proceso P7 que tiene prioridad 12, como esa prioridad es mayor que la del proceso que estaba ejecutando (P_1 con prioridad 10), el planificador decide que deberá desalojar a P_1 en beneficio de P_7, le comunica su decisión al despachador que deberá hacer el cambio de contexto, o sea, guardar los valores de los registros de la CPU en el PCB del procesos P_1, insertar

ese PCB al inicio de la cola de listos (se ha supuesto que la cola está ordenada por prioridades, pero no tiene que ser así), cargar la CPU con los valores del PCB del proceso P_7 y entregarle el control de la CPU.

Este algoritmo somete a largas esperas a los procesos con prioridades bajas. Cuando un proceso sufre esas esperas prolongadas se dice que está en inanición (*starvation*), una solución a ese problema es que el SO suba las prioridades a los procesos que llevan mucho tiempo en el sistema, de forma que en un futuro más cercano les toque el procesador. Se dice que los procesos se ganan el derecho a ejecutar porque "han envejecido" (Lezcano Brito, 2018).

Algoritmo SJF - El trabajo más corto primero

Este es un algoritmo que también usa prioridades, pero en este caso la prioridad viene dada por la longitud de la próxima ráfaga de CPU que necesitan los procesos. Se prioriza al proceso que tenga la próxima necesidad de CPU más corta, lo que se basa en el hecho de que el proceso usará ese recurso y lo liberará rápidamente. Ofrece el tiempo mínimo promedio para un conjunto de procesos dado (Hernández, et al., 2017).

Los estudios muestran que SJF arroja muy buenos resultados, pero es imposible predecir el futuro y por eso solo pueden implementarse algunas aproximaciones al algoritmo, por ejemplo, predecir ese tiempo como el promedio de las longitudes de los ciclos anteriores.

El planificador de periodo largo de los sistemas de tratamiento por lotes (batch) podría usar SJF, para lo cual es posible aprovechar el hecho de que en este tipo de sistema los usuarios someten los lotes acompañados por una descripción general y una especificación de cada proceso; en esta última, se puede incluir una estimación del tiempo (Lezcano Brito, 2018).

Algoritmo Round robin - Todos contra todos

El algoritmo *round robin*, está especialmente concebido para SO de tiempo compartido, es decir sistemas que comparten el tiempo de la CPU por períodos iguales que se denominan quantum de tiempos. Los procesos que arriban a la cola de listos entran al final de esa cola sin importar de dónde vengan (Lezcano Brito, 2018).

La idea del algoritmo es darle un tiempo igual a todos los procesos; una

vez transcurrido ese tiempo, se dice que ha expirado el tiempo, se le retira el procesador al proceso que tenía la CPU y se envía al final de la cola de listos (Lezcano Brito, 2018); después de lo cual se le entrega la CPU al primer proceso de esa cola, formándose una cola circular de procesos que entran y salen de la CPU (figura 16). Si un proceso termina antes del tiempo asignado o se bloquea por algún motivo (por ejemplo, solicita una E/S), simplemente abandona la CPU y entrará otro proceso antes de tiempo.

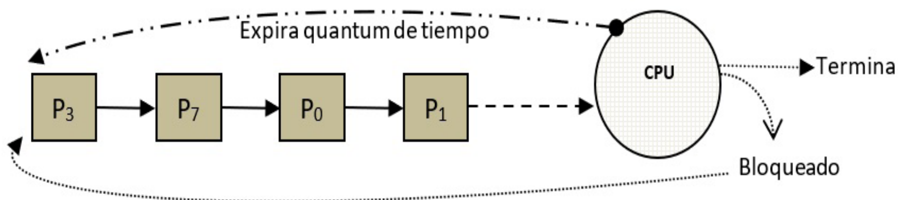


Figura 16. Esquematización de *Round Robin*.

Para controlar el tiempo, el sistema dispone de un temporizador que se inicia con el valor del quantum de tiempo cada vez que un proceso entra a la CPU. El temporizador disminuye su valor en uno cada milisegundo, de forma que expira cuando llega a cero (Lezcano Brito, 2018) y en ese momento entra en escena el despachador. Debe observarse que en este caso la labor del planificador de periodo corto es prácticamente nula.

Algoritmo de Colas múltiples

El algoritmo de colas múltiples posee varias colas de listos, en la figura 17 son dos.

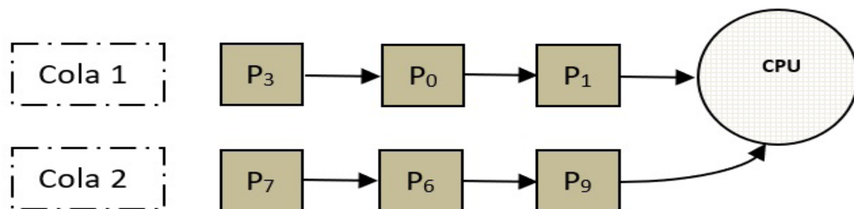


Figura 17. Colas múltiples.

Los procesos se asignan permanentemente a alguna de esas colas, para lo cual se toman en cuenta algunos aspectos, tales como: la cantidad

de memoria que necesitan, su prioridad y el tipo de proceso, entre otros. Cada cola tiene su propio algoritmo de planificación y existe un algoritmo general que decide cuál de las colas se atenderá (Lezcano Brito, 2018).

Algoritmo de Colas multinivel con retroalimentación

Este algoritmo también utiliza varias colas, pero a diferencia del anterior, los procesos pueden moverse entre ellas. A cada cola se asocia una prioridad que va desde la más prioritaria (la primera) hasta la menos prioritaria (la última). Los procesos que llegan a competir por la CPU entran a una cola en dependencia de su prioridad, en ella reciben un tiempo de CPU que es el menor de todos los que recibirá en las colas inferiores. Si un proceso P no termina en ese tiempo, se pasa a una cola inferior, en la cual recibirá un tiempo mayor de CPU, de modo que, según vaya migrando de arriba (Lezcano Brito, 2018) hacia abajo, se irá incrementando el tiempo de CPU hasta la última cola que se atiende FCFS, en donde se le da todo el tiempo que necesite en cada etapa (no se desaloja).

El algoritmo de planificación se hace de manera tal que solo atienda una cola inferior si las colas superiores están vacías. Si se está atendiendo una cola inferior y llegan procesos a las colas superiores, se desaloja el proceso de la cola inferior y se pasa a atender el de la cola superior. Para tratar de resolver el problema de inanición, se permite que el planificador ascienda a colas superiores a los procesos que llevan demasiado tiempo en el sistema (han envejecido) (Lezcano Brito, 2018).

1.4. Control de procesos

Como ya se analizó el estado de los procesos define su actividad actual y el SO es responsable de controlar su ejecución, para hacerlo deberá asignarle el procesador en diferentes momentos. Ese control se establece desde el momento en que el proceso nace (se crea) hasta que termina.

Creación de procesos

El SO es el responsable de crear los procesos; para hacerlo primero le construye un PCB, que será su CPU virtual, y después le gestiona algún espacio de memoria para cargarlo parcial o totalmente.

Los procesos pueden crearse debido a diferentes causas o acciones, las cuales pueden ser las siguientes:

- Por la acción de un usuario; por ejemplo, si se oprime el botón del ratón dos veces sobre una aplicación en un ambiente gráfico.
- Al ejecutarse una acción de un proceso; por ejemplo, la ejecución de una llamada al sistema fork() en los SO tipo UNIX.
- El SO crea el proceso a nombre de una aplicación, como sucede cuando se da la orden de enviar algo hacia un equipo de salida y el SO crea un proceso para manejar dicho equipo.

La figura 18 muestra un programa que usa la llamada al sistema fork() para crear un proceso pesado, que será hijo del proceso que lo creó.

```
# include<stdio.h>
# include<sys/types.h>
# include<unistd.h>
main ()
  {
    pid_t child;
    child = fork ();
    if (child < 0)
      {fprintf(stderr, "\nError al crear el proceso\n"); }
    if (child == 0)              //Código del hijo
      {
        printf ("\nSoy un proceso nuevo. Aquí debería hacer algo útil");
      }
    else        // Código del padre
      {
        printf ("\nSoy el proceso padre, esperaré por mi hijo");
        wait ();
      }
    exit (0); //Código común
  }
```


Activar Wind

Figura 18. Creación de procesos en UNIX.

Es importante comprender que la sentencia child = fork(); se evalúa de derecha a izquierda, es decir primero se ejecuta fork() que crea un proceso (Lezcano Brito, 2018) pesado, denominado hijo, después se hace la asignación a dos variables con el mismo nombre, child, pero en espacios de direcciones diferentes: una en el espacio del hijo y otra en el espacio del padre.

La llamada al sistema fork() crea un proceso nuevo y pesado, devolviendo: cero en el hijo y el identificador del proceso creado (su pid) en el padre. En caso de error devuelve -1.

El SO necesita una cierta cantidad de información para poder manipular procesos y recursos, dicho información se mantiene en diversas tablas que reflejan el estado de los recursos que manipula, en general existan tablas para:

- El manejo de la memoria: estas tablas incluyen información acerca de la localización de instrucciones y datos de cada proceso, tanto en memoria principal como secundaria o externa; los atributos de protección; información para manejar la memoria virtual, especificando si se comparte con otros procesos.
- Los equipos de entrada/salida. Esta información permite manejar los equipos y canales de E/S del sistema de cómputo (si están o no asignados, a quiénes están asignados, entre otros.), determina si se están usando como fuente o destino, así como las localizaciones de memoria desde donde proceden o se envían los datos.
- Los archivos. Permiten conocer la existencia de los archivos abiertos, su localización en memoria externa, su estado, entre otros.
- Los procesos. La tabla de procesos deberá enlazarse con las anteriores de alguna manera debido a que los procesos están almacenados en memoria, hacen E/S a través de los periféricos y trabajan con distintos archivos.

Un proceso es una entidad propietaria de recursos y también una unidad de planificación y despacho en el sentido que el SO le asigna el procesador cada cierto tiempo (lo despacha) para lo cual sigue alguna política de planificación.

Con estos nuevos detalles acerca de los procesos debe redefinirse el concepto:

Un proceso es un programa en ejecución que constituye una unidad propietaria de recursos que es planificada y despachada por el SO para usar el procesador.

Debe quedar claro que la manifestación física de un proceso puede verse como una entidad que contiene suficiente memoria para almacenar:

- Un conjunto de instrucciones a ejecutar (el programa).
- Un conjunto de datos.

- Una pila para controlar las llamadas a procedimientos y el paso de parámetros entre esos procedimientos.

El lugar de memoria donde se almacena el proceso depende de la técnica de manejo de memoria que usa el SO, ese tema se trata en el capítulo II.

1.5. Tipos de hilos

Como ya se analizó los hilos o procesos ligeros siempre son hijos de algún otro proceso, todos los hilos comparten un espacio de memoria y algunos otros elementos con su padre y sus hermanos. Los procesos pesados también pueden ser hijos de otros procesos, pero no comparten nada entre sí. Existen dos tipos de hilos: a nivel de usuario y a nivel de *kernel*.

Hilos a nivel de usuario

El *kernel* no conoce la existencia de este tipo de hilo que es manejado por la aplicación que los crea. Al nivel del SO y de algunos lenguajes de programación existen bibliotecas de hilos que contienen funciones para: crearlos, destruirlos, pasar mensajes, planificarlos, guardar su estado y restaurarlos.

Por defecto las aplicaciones comienzan con un proceso que solo contiene un hilo de ejecución, ese hilo lo maneja el *kernel*. Las aplicaciones pueden crear hilos en cualquier momento y ellos se ejecutarán junto al proceso que los creó.

Una biblioteca de hilos de usuarios contiene la funcionalidad necesaria para construir la estructura de datos que necesita cada hilo y también para planificarlos, crearlos, destruirlos y realizar el cambio de contexto entre los hilos y su proceso padre; en estas decisiones no participa el *kernel*. Por lo demás la vida de estos hilos transcurre igual que la de los procesos.

En la figura 19 se aprecia la manera en que interactúa un proceso padre con sus hilos a nivel de usuario. En este caso el proceso A tiene solo dos hilos (pudieran ser más): A1, y A2 que están listos para ejecutar. El proceso A es dueño del procesador o sea está en estado de ejecución:

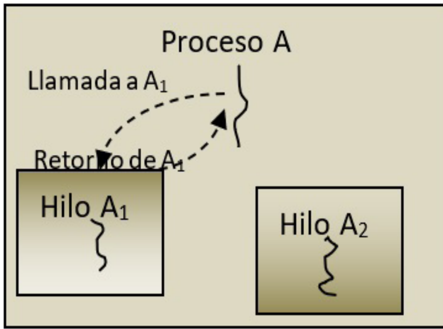


Figura 19. Paso de control entre un proceso y sus hilos.

En un instante dado la secuencia de ejecución del proceso A pasa al hilo A1, a pesar de eso A continuará en estado de ejecución, pero ahora su hijo A1 es el que tiene el control de la CPU y por eso el estado de A1 pasa de listo a en ejecución.

Cuando A1 termina pasa de nuevo al estado listo (podría ser llamado de nuevo), debe observarse que A permanecerá todo ese tiempo en el estado de ejecución porque tanto el código de A1 como el de A2 forman parte de él. Todos los cambios de contexto (desde A hacia A1 y viceversa o entre A y A2) se hacen a nivel de hilo y el *kernel* no interviene.

El *kernel* de los SO tipo UNIX tradicionales y las versiones más antiguas de Linux no ofrecen soporte para el trabajo multihilo, por eso es necesario usar alguna biblioteca de hilos a nivel de usuario; por ejemplo, Pthread (POSIX thread, Portable Operating Systems Interface, API IEEE estándar), la cual mapea todos los hilos de un proceso dado sobre un hilo simple a nivel de *kernel*.

La figura 20 muestra la idea esquematizada de esta concepción, la aplicación hace todo el trabajo y comienza ejecutando un hilo simple; ambos (la aplicación y el hilo) se asignan a un proceso simple, P, que maneja el *kernel*.

En cualquier momento, P puede crear hilos nuevos (H_1, H_2,...) para que ejecuten juntos con él. La biblioteca de hilos será la responsable de crear la estructura de datos necesaria para controlar el hilo, una vez que se cree el hilo el control pasa a P, cuando el control pase del hilo a la biblioteca se guarda su contexto y cuando sea a la inversa se

restaura.

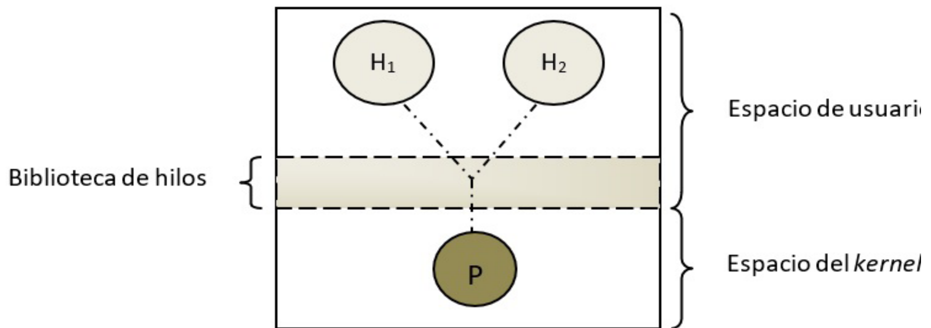


Figura 20. Proceso pesado con dos hilos a nivel de usuario.

Creando un hilo con Pthread

El análisis completo de la biblioteca Pthread está fuera del alcance de este texto, solo se presenta un ejemplo simple para comprender los aspectos analizados.

```
#include <pthread.h>                                   //Usar biblioteca Pthread
#include <stdio.h>                                     //Usar la biblioteca de entrada/salida estándar

void *function(void *arg)                              //Función que invocará el hilo creado
   {
       printf("Estoy en el hilo");                     //Aquí se haría algo útil
       return 0;//El hilo termina y devuelve el control a su padre
   }

int main(void) //Este es el padre
   {
       pthread_create(&tid, NULL, function, NULL);     //El hilo tid ejecuta la función function
       pthread_join(tid, NULL);                        //El padre espera por su hijo tid
           return 0;
   }
```

Figura 21. Proceso pesado con un hilo a nivel de usuario.

La biblioteca Pthread incluye un conjunto de funciones que permiten trabajar con hilos. La figura 15 muestra un programa de ejemplo que utiliza las funciones:

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void*), void *arg) (Lezcano Brito, 2018), todos sus parámetros son apuntadores y tienen el significado siguiente: thread es el

identificador del hilo creado, attr contiene los atributos del hilo (si es NULL se usarán los atributos por defecto), start_routine() es la función que ejecutará el hilo, arg contiene los argumentos usados por start_routine().

- int pthread_join(pthread_t thread, void **value_ptr), el argumento thread identifica el hilo por el que debe esperarse (si ya terminó no tiene efecto) y value_ptr contiene el código de terminación del hilo por el que se espera.

Hilos a nivel de kernel

Cuando se usa este tipo de hilos, el *kernel* hace el trabajo y por eso en la aplicación no hay código para manejarlos, una API sirve de interfaz con el *kernel* que se encarga de hacer todo.

Esta implementación está presente en la biblioteca Win32 de los SO Windows, en este caso la planificación se hace a nivel de hilos y para el SO no existen diferencias entre un hilo y un proceso pesado, lo cual permite que se puedan planificar varios hilos de un mismo proceso que pueden ejecutar sobre diferentes procesadores (cuando existan), por otra parte, cuando un hilo de un proceso se bloquea es posible planificar otro hilo de ese mismo proceso.

En esta implementación las rutinas del *kernel* también pueden ejecutarse como hilos múltiples, la principal desventaja es que se hace necesario cambiar el modo de ejecución (hacia el modo *kernel*) para realizar el cambio de contexto entre dos hilos de un mismo proceso.

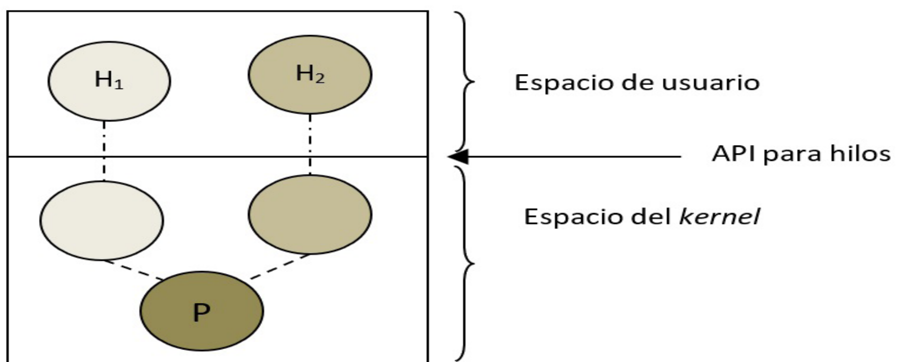


Figura 22. Proceso pesado con dos hilos a nivel del *kernel*.

La figura 22 muestra la idea esquematizada de esta concepción, tanto el proceso P como sus hilos H1 y H2 están controlados por el *kernel* y la API sirve de interfaz con él.

Implementaciones combinadas

Algunos SO usan una combinación de los dos casos analizados anteriormente, en esa implementación todos los hilos de la aplicación se crean, como un conjunto, en el espacio de usuario. Los hilos de una aplicación se mapean sobre hilos de *kernel* (el programador puede ajustar cómo). El SO Solaris usa esta forma de hilos.

1.6. Concurrencia

La concurrencia está relacionada con la ejecución simultánea de varios procesos. Es un procesado concurrente (o procesado paralelo) la circunstancia en la que, de tomar una instantánea del sistema en conjunto, varios procesos se vean en un estado intermedio entre su estado inicial y final (Correia Barbosa, 2019).

La multiprogramación se ideó para permitir que varios procesos puedan ejecutar en un medio con un solo procesador, para hacerlo es necesario que los procesos compartan recursos que se asignan y liberan dinámicamente, lo cual trae consigo condiciones de competencia.

Los programadores pueden hacer aplicaciones que están constituidas por un conjunto de procesos que cooperan entre sí o sea actúan de forma concurrente, p muchos SO se diseñan e implementan como un conjunto de procesos o hilos concurrentes.

Aunque la concurrencia se puede ver tanto en los SO multiprogramados como en los de multiprocesamiento y los distribuidos y existen bastante similitudes en la forma de abordarla, también existen suficientes diferencias para tratar estos aspectos de forma independiente. En este texto solo se aborda la concurrencia desde el punto de vista de la multiprogramación.

Principios de concurrencia

Como ya se ha apuntado, los SO multiprogramados se caracterizan por ejecutar procesos que se intercalan en el tiempo dando la sensación de que realizan su trabajo a la vez. Los SO de multiprocesamiento permiten

la ejecución en paralelo, o sea al mismo tiempo, de tantos procesos como procesadores centrales existan.

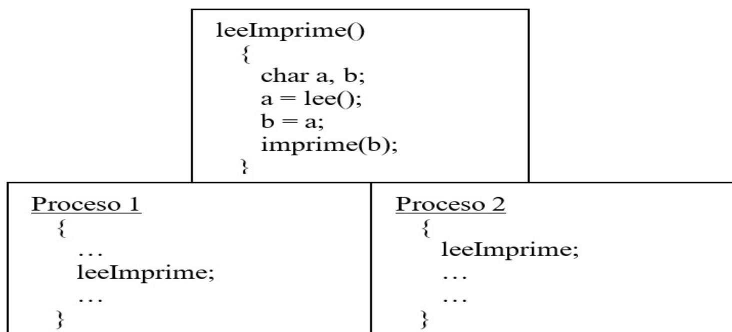


Figura 23. Dos procesos concurrentes.

La figura 23, muestra uno de los problemas a los que se enfrentan los procesos concurrentes. Dos procesos comparten la función leeImprime(), en un instante dado el Proceso 1, invoca al procedimiento compartido, logra leer el carácter

tecleado por el usuario y lo almacena en la variable a, pero el SO le retira el procesador para entregárselo al Proceso 2, el cual invoca la función leeImprime() que se ejecuta sin interrupción, leyendo e imprimiendo un carácter nuevo y perdiéndose el valor que tenía la variable a, que era válido para el Proceso 1, el cual retorna en algún momento y realiza la operación que tenía pendiente dentro de la función leeImprime(), como el valor que tiene la variable a es el leído por el Proceso 2 se imprimirá un valor incorrecto.

La concurrencia en los SO multiprogramados y los de multiprocesamiento presentan los mismos problemas:

- Al compartir los recursos pueden ocurrir errores pocos predecibles, como el visto anteriormente donde se imprime un valor incorrecto.
- Se le hace difícil al SO asignar los recursos dinámicamente.
- Resulta muy difícil localizar los errores de programación debido al comportamiento no predecible del orden de ejecución. Obsérvese que el ejemplo que se presentó en la figura 17 depende de los momentos en que se intenta invocar la función leeImprime(), los dos procesos pudieran

ejecutarse muchas veces sin producirse errores, no obstante no se garantiza que sea así siempre.

Condiciones de Bernstein

Las condiciones de Bernstein (1966), establecen los requisitos que tienen que cumplir dos conjuntos de sentencias para que puedan ejecutarse de forma concurrente. Antes de definirlas se establece la notación siguiente:

- E(Ck) es el conjunto formado por todas las variables de entrada que intervienen en la ejecución de una secuencia de instrucciones Ck, entendiéndose por variables de entradas aquellas que se usan sin alterar sus valores.
- S(Ck) es el conjunto de todas las variables de salida que participan en la ejecución de una secuencia de instrucciones Ck, se designan variables de salida a todas las que cambian su valor.

Bernstein establece que dos conjuntos de sentencias, C_i y C_j, se puedan ejecutar concurrentemente si cumplen las siguientes condiciones:

$$E(C_i) \cap S(C_j) = \emptyset$$
$$E(C_j) \cap S(C_i) = \emptyset$$
$$S(C_i) \cap S(C_j) = \emptyset$$

Figura 24. Condiciones de Bernstein.

Si se cumplen esas tres condiciones, se dice que los conjuntos no tienen una relación de dependencia entre sus variables. Debe observarse que (Lezcano Brito, 2018):

- Si se incumple la primera condición, no existe garantía del valor que tendrían las variables cuando se esté ejecutando la secuencia de sentencias C_i, dado que pueden ser alteradas por la secuencia de sentencias C_j que se está ejecutando concurrentemente con C_i. Es decir, el valor que se obtenga en Ci es dependiente de C_j debido a que Cj puede alterar los valores que leerá C_i.
- Para la segunda condición sucede el mismo problema, pero ahora las variables pueden cambiar su valor en C_i, lo cual puede afectar a C_j.
- Por último, es fácil notar que existe una dependencia mutua entre C_i y C_j cuando se incumple la tercera condición, ya que en ambas secuencias

de sentencias se alteran los valores de las variables compartidas (Lezcano Brito, 2018).

Queda claro que no existe ningún problema cuando $E(S_i) \cap E(S_j) \neq \emptyset$, debido a que en este caso ambas secuencias de sentencias están usando los valores de las variables (Lezcano Brito, 2018), pero no los están alterando.

Debe observarse que puede que no haya problema, porque estos estarán relacionados con el orden en que se ejecuten las sentencias, pero eso es impredecible y por tanto no existe garantía alguna de que siempre se alcance el resultado esperado. Las condiciones de competencia ocurren cuando dos o más procesos leen y escriben datos de forma dependiente lo que puede afectar los resultados obtenidos.

Sección crítica

Para dos o más procesos que comparten variables comunes, la sección crítica (SC) está formada por el conjunto de sentencias que violan las condiciones de Bernstein (Lezcano Brito, 2018).

Por supuesto que dos procesos que no comparten nada no tendrán secciones críticas entre ellos. Por el contrario, si dos procesos comparten algunas variables, no puede ser que uno lea y otro escriba sobre ellas o que ambos escriban sobre ellas sin tomar ningún tipo de cuidado (Lezcano Brito, 2018).

Lo importante en este esquema es que las secciones críticas de ambos procesos se ejecuten en forma excluyente; es decir, de entre varios procesos que cooperan entre sí de forma concurrente solo uno de ellos puede estar en cada instante de tiempo dentro de su correspondiente sección crítica (Lezcano Brito, 2018).

Para trabajar con procesos concurrentes es necesario identificar las secciones críticas y tomar cuidado al usarlas. Con ese propósito se establecen protocolos de entrada y de salida a esas secciones de manera que la ejecución del código crítico se haga de manera excluyente. La idea esquematizada se observa en la figura 25.

Proceso P	Proceso Q
Código no crítico	Protocolo de entrada
…	SC
Protocolo de entrada	Protocolo de salida
SC	Código no crítico
Protocolo de salida	…
Código no crítico	…
…	…
…	…

Figura 25. Proceso pesado con dos hilos a nivel del *kernel*.

Obsérvese que no puede asumirse nada con relación a la velocidad de los procesos ni al momento en que un proceso entrará en su SC (por la posición que ocupe o por cualquier otra razón).

Cualquier solución al problema de la sección crítica para un conjunto de procesos, P = {p1, p2,..., pn}, que trabaja en forma concurrente debe satisfacer los siguientes requisitos (Lezcano Brito, 2018):

1. Exclusión mutua. Solo un proceso pi del conjunto P, puede estar ejecutando su sección crítica SCi. Durante este tiempo si algún otro proceso pj, j ≠ i, desea ejecutar su sección crítica SCj, tendrá que esperar a que pi termine.
2. Progreso. Solo los procesos interesados en entrar en su respectiva sección crítica pueden participar en la decisión de cuál lo hará. Esto lleva implícito, por ejemplo, que no se puedan establecer turnos (Lezcano Brito, 2018).
3. Espera limitada. Después que un proceso pi haya hecho una solicitud para entrar en su sección crítica SCi, debe existir un límite de veces que se le permita a otros procesos hacerlo. Esto significa que ningún proceso debe esperar indefinidamente (lo contrario provocaría inanición) para entrar a su sección crítica por la razón que sea (prioridad, por ejemplo).

Solución de Peterson para la sección crítica

Esta es una solución, por software, que resulta bastante sencilla y funciona bien en arquitecturas de computadoras antiguas, pero no en algunas modernas; no obstante, se analiza en este apartado debido a

su sencillez.

La solución se restringe a dos procesos (P0 y P1) que comparten las variables turn y flag (figura 26) (Lezcano Brito, 2018):

- La variable turn, de tipo int (entero) se usa para hacer una doble verificación a la entrada de SC.
- El arreglo booleano, de dos elementos, flag se utiliza para indicar si un proceso i está interesado en entrar en su sección crítica.

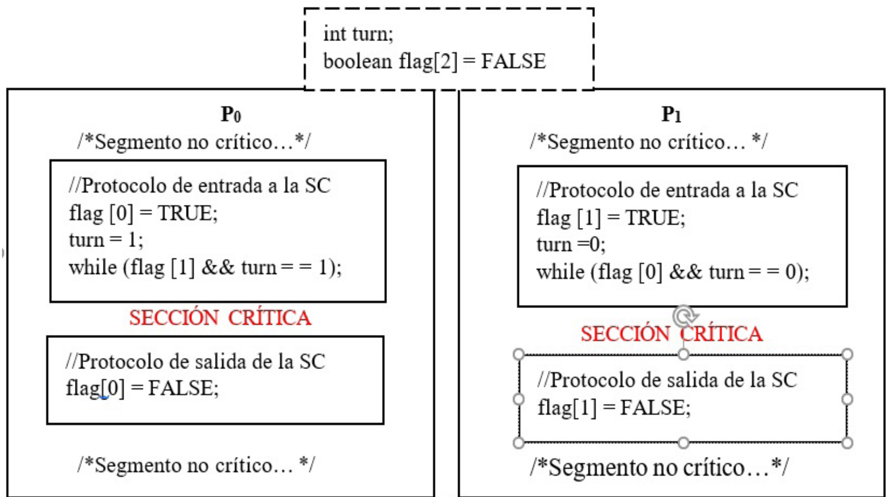


Figura 26. Sincronización de procesos usando la solución de Peterson.

La solución de Peterson cumple las tres exigencias que debe satisfacer una solución al problema de la sección crítica, obsérvese el siguiente análisis (Lezcano Brito, 2018):

Para probar la exclusión mutua, se puede observar que los procesos entran en su sección crítica bajo las siguientes circunstancias (Lezcano Brito, 2018):

- Cuando solamente un proceso está interesado en entrar le asigna a su índice dentro del arreglo flag el valor TRUE, mientras que el índice del otro permanece FALSE. Por ejemplo si flag[0] es TRUE y flag[1] es FALSE, P0 quiere entrar en su SC y P1 no. En esta situación el protocolo de entrada a la SC de P0 ejecuta la sentencia while(flag[1] && turn == 1); que se evaluará como FALSE y P0 entrará en la SC sin problemas.

- Si P1 y P0 están interesados en entrar a sus secciones críticas, ambos le asignarán TRUE a su índice del arreglo flag y después le asignarán valor a la variable turn (P0 asigna 1 y P1 asigna 0), quedando dicha variable con el valor asignado por el último proceso, el que quedará en una espera ocupada en su ciclo while.
- La espera ocupada ocurre cuando un proceso espera por una condición probándola constantemente.
- Para probar el progreso, basta con observar que los procesos que no desean entrar a la sección crítica tienen la variable flag[i] (i = 0, 1) con el valor FALSE y, por tanto, no toman parte en la decisión de quién entra.
- La espera limitada se ve claramente al observar que son solo dos procesos, y si ambos piden permiso a la vez, el que no entre quedará probando el ciclo while, del cual saldrá inmediatamente después que el otro proceso cambie el valor de la variable flag en su protocolo de salida de la SC, o sea en esta situación los procesos se alternan.

Soluciones por hardware

El análisis de las soluciones por *hardware* puede ser bastante sencillo debido a que ellas tienen la garantía de hacerse de manera atómica lo que significa que una vez iniciadas no se pueden interrumpir y solo un proceso la puede ejecutar en un instante de tiempo dado. Solo se analizarán algunas soluciones, pero existen otras.

Deshabilitar las interrupciones

En sistemas con un solo procesador los procesos ejecutan hasta que invocan un servicio del SO o son interrumpidos, por eso si se deshabilitan las interrupciones antes de que un proceso entre en su SC y se vuelven a habilitar cuando salga, se logra la exclusión mutua que es el objetivo inviolable para trabajar con procesos concurrentes.

La capacidad para hacer lo anterior la puede proporcionar el SO en forma de primitivas que se definen en el núcleo del SO. El protocolo para cada proceso consiste de tres pasos:

1. Deshabilitar las interrupciones antes de la SC.
2. Ejecutar el código crítico.
3. Habilitar las interrupciones al terminar la parte crítica; la idea se muestra en la figura 27.

```
                 Proceso_i
/*****************
* Código no crítico**
*****************/
Deshabilitar interrupciones
SC
Habilitar interrupciones
/*****************
* Código no crítico**
*****************/
```

Figura 27. Sincronización de procesos deshabilitando interrupciones.

Aunque esta solución funciona, el precio se paga con la disminución de la eficiencia, además no es aplicable en sistemas con varios procesadores debido a que deshabilitar las interrupciones en todos los procesadores no es algo tan sencillo y puede traer diversos inconvenientes.

Instrucciones de máquina especiales: compareAndSwap

Antes de ver los detalles debe quedar claro que la instrucción compareAndSwap se hace de manera ininterrumpida por un solo proceso (el *hardware* garantiza que así sea).

```
int compareAndSwap ( int *compare, int test, int new)
  {
    int old;
    old = *compare;
    if (old == test)
      *compare = new;
    return old;
  }
```

Figura 28. Definición de compareAndSwap.

La figura 28, muestra la definición de esta instrucción, obsérvese que en la sentencia if se compara el contenido apuntado por compare con el valor de test, si son iguales se reemplaza la localización apuntada por compare con el valor de new. Siempre se retorna el valor de old, de esta forma solo se actualiza el valor de la memoria si compare y test son iguales.

Obsérvese que si esa instrucción se hiciera por *software*, el código de la figura 21 no sería útil porque todas las variables mencionadas serían críticas. Algunas versiones de esta solución se encuentran en la familia de procesadores: x86, IA64, sparc, entre otros.

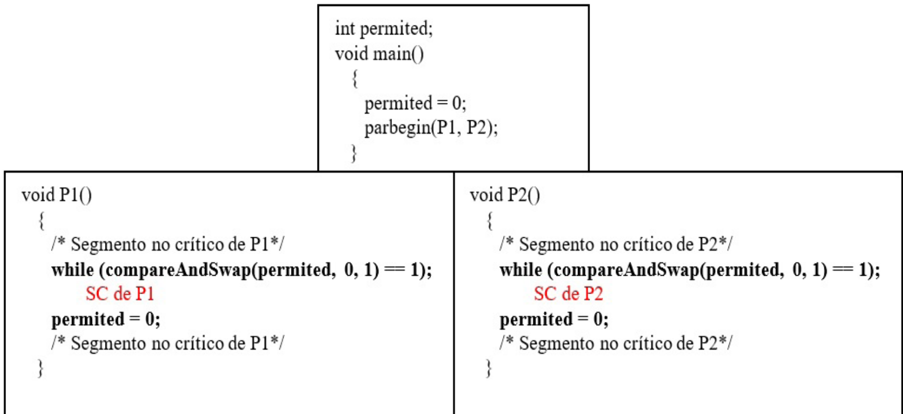


Figura 29. Uso de compare&swap() para sincronizar procesos.

La figura 29 muestra el uso de esta función, en la parte común se declara la variable compartida permited que se inicializa en 0, después se ordena la ejecución de dos procesos en forma concurrente, utilizando la sentencia parbegin(P1 y P2). Los procesos P1 y P2 tienen códigos diferentes, pero utilizan recursos comunes que violan las condiciones de Bernstein y por eso tienen secciones críticas (SC) en aquellos lugares donde se violen dichas condiciones, de manera que deberán tomar cuidado para ejecutar esos segmentos de código. Si varios procesos intentan entrar a su respectiva SC se encontrarán con el código while(compareAndSwap(permited, 0, 1) == 1); solo uno podrá ejecutar la instrucción compareAndSwap() y lo hará de manera atómica, cuando finalice de ejecutarla la variable permited tendrá el valor 1; los próximos procesos que puedan ejecutar compareAndSwap() (uno cada vez) se quedarán en una espera ocupada hasta que el proceso que está dentro de la SC salga de ella y le cambie el valor a permited (permited = 0;). En la figura se han resaltado en negritas los protocolos de entrada y de salida de las secciones críticas.

Semáforos

Los semáforos constituyen una solución proporcionada por el SO, la idea fue concebida por Dijkstra quien fue un científico holandés, premio Turing en 1972 que hizo notables aportes a la computación y se usó por primera vez en el SO THEOS que es un sistema operativo que originalmente se llamó OASIS (Lezcano Brito, 2018). Se basa en el principio

de que dos o más procesos pueden cooperar si se envían señales a través de variables especiales, llamadas semáforos.

Las señales se manejan usando dos primitivas nombradas signal() y wait(); el semáforo en sí es una variable especial sobre la que solo se pueden hacer tres operaciones: inicializarla con un valor no negativo y efectuar las operaciones atómicas wait() y signal().

La figura 30 muestra una definición de semáforo en la cual la variable mencionada anteriormente es del tipo struc semaphore y consta de dos campos:

- value, usada para cerrar o abrir el semáforo;
- list, es una cola de procesos esperando por el semáforo.

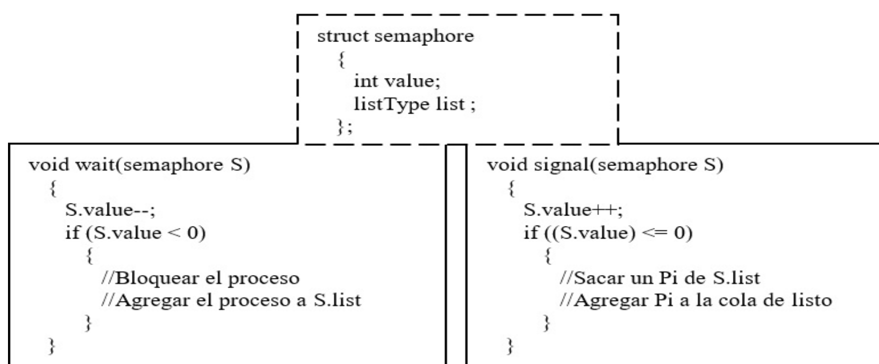


Figura 30. Definición de semáforo.

La primitiva wait(S) comienza restándole 1 a S.value si el valor no es negativo se ejecuta hasta el final, si es negativo bloquea al proceso solicitante y lo pone en la cola S.list donde deberá esperar hasta que se abra el semáforo (no hay espera ocupada).

La primitiva signal(S) incrementa el valor de S.value, si el resultado es menor o igual a cero habrá algún proceso esperando por el semáforo, por eso extrae uno de ellos de la cola S.list y lo incorpora a la cola de listos donde deberá competir por el procesador.

Existen tres tipos de semáforos: binarios, mutex y contadores:

- Los binarios pueden inicializarse con los valores 0 y 1.

- Los mutex constituyen un tipo especial de semáforo binario que exige ser cerrado por el mismo proceso que lo abre (los binarios no tiene esa restricción).
- Cuando los semáforos binarios o los mutex están cerrados no se puede usar el recurso controlado por ellos.
- Los contadores permiten controlar las instancias disponibles de un recurso, para lo cual deben inicializarse con la cantidad total de ese tipo de recurso.

Un problema de concurrencia típico

El problema del productor-consumidor se usa como ejemplo en la mayoría de los libros y cursos de SO y se basa en la existencia de dos procesos que cooperan entre sí, denominados productores y consumidores:

- Los productores producen algo y lo depositan en un búfer compartido.
- Los consumidores toman los elementos del búfer.

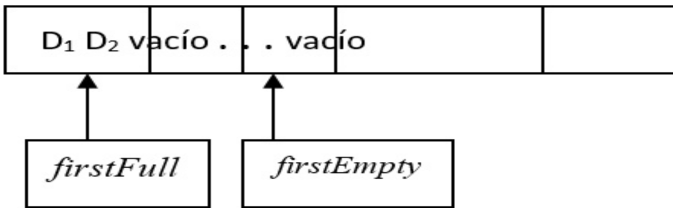


Figura 31. Búfer acotado a m posiciones para el productor-consumidor.

El búfer compartido se muestra en la figura 31, a él se asocian dos punteros: firstFull, apunta al primer elemento a consumir y firstEmpty, apunta al primer espacio libre del búfer.

El búfer que se usa es circular:

- Está vacío cuando firstFull==firstEmpty.
- Está lleno cuando ((firstEmpty + 1) % n) == firstEmpty.

Debe tomarse en cuenta que el productor no puede agregar datos cuando el búfer está lleno y el consumidor no puede consumir cuando el búfer está vacío. El código para el problema descrito se aprecia en la figura 32.

```
#define LONG = 100
typedef struct
    {
        ...
    } data
item buf [LONG];
int firstFull      = 0;
int firstEmpty = 0;
```

```
void producer()
    {
        data newData;
        while(true)
            {
                makeData(newData);
                while ((firstEmpty + 1) % LONG == firstFull)
                    ; // Espera a que haya espacio
                buf[firstEmpty] = newData;
                firstEmpty = (firstEmpty + 1) % LONG;
            }
    }
```

```
void consumer()
    {
        while(true)
            {
                while ((firstEmpty == firstFull)
                    ; // Espera a que haya datos
                consume(buf[firstFull]);
                firstFull = (firstFull + 1) % LONG;
            }
    }
```

Figura 32. Solución al problema productor-consumidor.

Aunque pueden existir varios productores y consumidores solo se analiza el caso de dos procesos:

- El productor construye un dato, makeData(newData). Si no hay espacio, (firstEmpty + 1) % LONG == firstFull, queda en una espera ocupada (ciclo while). Si hay espacio coloca el dato, buf[firstEmpty] = newData, y desplaza el puntero a la primera posición vacía, firstEmpty = (firstEmpty+1) % LONG.
- El consumidor comprueba si no hay datos, firstEmpty == firstFull. Si no hay nada que consumir se queda en una espera ocupada (ciclo while). Si hay datos, procesa el primero, consume(buf[firstFull]), y desplaza el puntero a los datos, firstFull = (firstFull + 1) % LONG.

La solución presentada solo permite usar LONG - 1 localizaciones debido a la forma en que se mueve el puntero firstEmpty: firstEmpty=(firstEmpty + 1) % LONG. Obsérvese que si el puntero está en la penúltima posición del búfer, la parte derecha de la ecuación se avaluará como: (LONG - 1 + 1) % LONG moviendo el puntero a la localización 0, lo que deja sin usar la localización LONG.

Segunda solución

Con el propósito de contar los elementos del búfer y usar su última posición se introduce la variable compartida counter (figura 33).

```
#define LONG = 100
typedef struct
   {
      ...
   } data
item buf [LONG];
int firstFull     = 0;
int firstEmpty   = 0;
```

```
void producer()
  {
    data newData;
    while(true)
      {
        makeData(newData);
        while (counter == LONG)
          ; // Espera a que haya espacio
        buf[firstEmpty] = newData;
        firstEmpty = (firstEmpty + 1) % LONG;
        counter++
```

```
void consumer()
  {
    while(true)
      {
        while ((counter == 0)
          ; // Espera a que haya datos
        consume(buf[firstFull]);
        firstFull   =   (firstFull   +   1)   %
LONG;
        conter--
      }
```

Figura 33. Problema productor-consumidor. Intento de usar todo el búfer.

Puede que al lector le resulte esta idea atractiva y un análisis ligero pudiera llevarle a la conclusión de que resuelve el problema, pero la realidad es más compleja debido a que counter se ha convertido en una variable crítica y este tipo de variable se relaciona directamente con las condiciones de competencia.

El problema ocurre cuando los procesos se ejecutan en forma concurrente debido a que unas veces se obtendrán resultados correctos y otras no (es una característica de los procesos concurrentes cuando hay errores). Antes de entrar en detalles debe señalarse que la traducción de counter++ y counter-- a código de máquina es la siguiente:

counter++	counter--
register1 = counter register1 = register1 + 1 counter = register1	register2 = counter register2 = register2 - 1 counter = register2

Figura 34. La traducción de counter++ y counter-- a código de máquina.
Fuente: Lezcano Brito (2018).

Con estas ideas en mente se presenta una situación en la cual se ha producido una cierta cantidad de datos, 12 para ejemplificar, o sea (counter es 12). El productor elabora un dato y ejecuta la sentencia counter++, pero solo logra hacer las dos primeras operaciones, o sea llega hasta: register1 = register1 + 1, donde se le retira el procesa-

dor, para asignárselo al consumidor (¡no se ha actualizado el valor de counter!, que debería ser igual a register1 o sea 13, pero el valor real es 12.

El consumidor consume algo y ejecuta la sentencia counter-- completa, como counter no está actualizado, obtiene el valor 11 que es erróneo y además, cuando el consumidor toma el control de nuevo, counter recibirá 13 porque ese fue el valor que el guardó de register1 que es es un registro hipotético de una máquina dada (o cualquier otro registro que se haya utilizado).

Como se puede ver aquí hay una inconsistencia muy grave, los resultados también podrían ser otros, de acuerdo a como se planifiquen los procesos, el valor real de counter debería ser 12 porque se produjo un dato y se consumió otro, así que la cantidad de datos se mantiene igual que al inicio (no así los punteros).

El problema que se ha descrito tiene que ver con el mal uso que se ha hecho de la variable counter, la cual viola las condiciones de Bernstein y por tanto es crítica.

Una solución al problema del productor-consumidor usando semáforos

En esta sección se presenta una nueva solución que utiliza tres semáforos:

- El semáforo s, de tipo mutex, para controlar el acceso exclusivo al búfer.
- El semáforo contador free para contar la cantidad de espacios libres. Inicialmente el búfer está vacío, o sea free = m.
- El semáforo contador full para contar la cantidad de espacios ocupados. Ninguno al inicio, full = 0.

El trabajo del productor es el siguiente (parte izquierda de la figura 28):

1. Prepara el dato, makeItem().
2. Comprueba si hay espacio, usando el semáforo contador free (wait(free)):
   - Si no hay espacio, (wait(free)) bloquea al productor y lo agrega a la cola asociada al semáforo free.
   - Si hay espacio comprueba si el semáforo s está abierto (wait(s)):

-Si s está cerrado, (wait(s)) bloquea al productor y lo agrega a la cola asociada al semáforo s.
-Si s está abierto, wait(s) lo cierra, deposita el dato (addItem()), abre el semáforo (signal(s)) e informa que hay un dato más signal(full).

```
void producer ()
  {
    while (true)
      {
        makeItem ();    //Prepara el dato
        wait(free);     //Si free < 0, búfer lleno
        wait(s);        //¿Puedo entra en SC?
        addItem ();     //Pone el ítem en el búfer
        signal(s);      //Avisa que salió de la SC
        signal(full);   //Hay otro ítem
      }
}
```

```
void consumer()
  {
    while (true)
      {
        wait(full);     //¿Hay algún ítem?
        wait(s);        // ¿Puedo entra en SC?
        takeItem();     //Extraer un ítem
        signal(s);      //Salida de la SC
        signal(free);   //Hay otro espacio
        consItem();     // Consume un ítem
      }
  }
```

```
const int m                          //Tamaño del búfer
binary semaphore     s = 1;          //Controla la SC
count semaphore      full = 0;       //Comienza con búfer vacío y…
                     free = m;       //… todo búfer libre

void main()
  {
    parbegin (producer, consumer);
  }
```

Activ

Figura 35. El problema del productor-consumidor con semáforos.

El consumidor hará las acciones siguientes (parte derecha de la figura 35):

1. Comprueba si hay algo en el búfer, usando el semáforo contador full (wait(full)):

- Si no hay datos, (wait(full)) bloquea al productor y lo agrega a la cola asociada al semáforo full.
- Si hay datos comprueba si el semáforo s está abierto (wait(s)):
  -Si s está cerrado, (wait(s)) bloquea al consumidor y lo agrega a la cola asociada al semáforo s.
  -Si s está abierto, wait(s) lo cierra, toma el dato (takeItem()), abre el semáforo (signal(s)), informa que hay un espacio nuevo (signal(free)) y consume el dato (consItem()).

Las funciones addItem() y takeItem() deberán actualizar los punteros al

primer espacio vacío y al primer espacio lleno del búfer respectivamente, ellas en sí son las SC.

Los monitores una solución a nivel de lenguaje de programación

Los semáforos constituyen una buena solución al problema de la sección crítica, pero si los programadores los usan en forma incorrecta pueden acarrear errores, algunos tan graves como el deadlock (se tratará más adelante).

Los monitores se ofertan en bibliotecas de programas de algunos lenguajes de programación; por ejemplo, Pascal concurrente, Modula y Java, con una funcionalidad equivalente a los semáforos que resulta fácil de controlar por los programadores.

Según la definición inicial, brindada por Hoare (1974), quien fue un notable científico británico que ha hecho valiosos aportes a la computación, un monitor es un módulo de *software* formado por: uno o varias funciones o procedimientos, una secuencia de inicialización y un conjunto de datos locales.

Asociado a cada monitor existe una cola de entrada de procesos compitiendo por el uso del monitor. Los monitores son datos de tipo abstractos que encapsulan contenidos solamente accesibles desde sus funciones internas (figura 36).

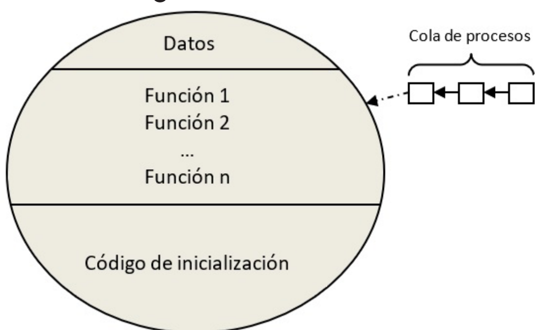


Figura 36. Vista esquemática de un monitor.

Los monitores se caracterizan por la siguiente funcionalidad:

- Sus variables locales solo pueden accederse desde los procedimientos del monitor.

- Los procesos deberán invocar uno de los procedimientos del monitor para entrar a él.
- Solo un proceso puede ejecutar el monitor en un intervalo de tiempo dado y mientras lo esté haciendo cualquier otro proceso que lo invoque quedará bloqueado, en espera de que el proceso que esté dentro termine (dejando disponible al monitor).

Obsérvese la familiaridad de las dos primeras características con el paradigma de la programación orientada a objetos tan popular hoy en día en la comunidad de los programadores.

La última característica ofrece la exclusión mutua que se exige para los problemas de las secciones críticas en procesos concurrentes, permitiendo que el acceso a las variables sea excluyente en el tiempo. Para el procesamiento concurrente los monitores incluyen herramientas de sincronización a través del uso de variables especiales, denominadas variables de condición, que se operan a través de dos funciones:

- cwait(c), suspende la ejecución del proceso que la invoca en espera de que se cumpla la condición c, liberando el monitor para que otro proceso lo pueda usar.
- csignal(c), reinicia algún proceso bloqueado por la condición c. Si no hay proceso esperando esa condición no hace nada.

La figura 37 muestra la sintaxis para definir monitores.

```
monitor nombreDelMonitor
  {
    // Declaración de variables compartidas
       …
    // Definición de las funciones del monitor
       f1() {}
       f2() {}
       ...
    // Código de inicialización
       initialization_code() {}
  }
```

Figura 37. Sintaxis para definir monitores.

La figura 38 retoma el problema del productor-consumidor para ejemplificar el uso de los monitores.

```
monitor productorConsumidor                        //Define un monitor denominado productorConsumidor
  {
    /** Variables locales al monitor **/
    char buffer [M];                               //Búfer de tamaño M
    int in, out;                                   //Apuntadores al búfer (in, entrada; out salida)
    int count;                                     //Cantidad de elementos en el búfer
    cond freeSpace, avaibleItem;                   //Variables de condición

    /** Procedimientos del monitor **/
    void add(char x)                               //Agregar x al búfer. Lo usa el productor
      {
        if (count == M)                            //El búfer está lleno
          cwait(freeSpace);                        //El proceso productor debe esperar
      buffer[in] = x;                              //Agrega lo producido en x al búfer
        in = (in + 1) % M;                         //Mueve el puntero a la próxima localización vacía
        count++;                                   //Hay un elemento más en el búfer
        csignal (avaibleItem);                     //Reinicia un consumidor, si ninguno espera no tiene efecto
      }
    char get()                                     //Extrae un dato x del búfer. Lo usa el consumidor
      {
        if (count == 0)                            //El búfer está vacío
          cwait(avaibleItem);                      //El proceso consumidor debe esperar
        x = buffer[out];                           //Extraer un elemento del búfer
        out = (out + 1) % M);                      //Mueve el puntero a la próxima localización ocupada
        count--;                                   //Hay un elemento menos en el búfer
        csignal (freeSpace);                       //Reinicia un productor, si ninguno espera no tiene efecto
        return x;                                  //Devuelve el dato extraído del búfer
      }
    /** Código de inicialización del monitor **/
    initialization_code ()
      {
        in = 0; out = 0; count = 0;                // Inicializar el búfer vacío
      }
  }
```

Figura 38. Problema del productor-consumidor.

Es importante resaltar que se puede entrar a un monitor invocando cualquiera de sus procedimientos, pero una vez que cualquier proceso entra a él cierra todas las entradas.

El monitor definido, productorConsumidor, tiene la responsabilidad de controlar el uso del búfer acotado buffer, que admite M elementos.

La solución incluye las siguientes variables adicionales, además de buffer:

- in para apuntar a la próxima entrada libre.
- out para apuntar al próximo elemento a consumir.
- Las variables de condición (declaradas con la palabra reservada cond):
  - freeSpace para indicar si hay espacio libre en el búfer. En ese caso el productor puede depositar lo producido.

- avaibleItem para indicar si hay algún elemento en el búfer. En esa situación el consumidor puede tomar un dato.

El monitor productorConsumidor que se ha definido incluye dos funciones:

- add(), que espera un dato de tipo char para agregarlo al búfer. Esta función sola la usa el productor, pero no por restricciones propias de los monitores sino por el tipo de problema que se está resolviendo.
- get(), que toma un dato del búfer. Esta función solo la utiliza el consumidor por las mismas razones de la función anterior.

La sección de inicialización es la responsable de inicializar el contador de elementos en cero y los punteros.

La figura 39 muestra los códigos del productor (parte izquierda) y del consumidor (parte derecha). La función main() los invoca, en forma concurrente, usando la función parbegin():

- El productor agrega el dato producido, invocando la función del monitor add(); la cual garantiza: la exclusión mutua, que haya espacio y que se reinicie algún consumidor bloqueado, si existe (usando una variable de condición).
- El consumidor obtiene un dato usando, la función del monitor get(); la cual garantiza: la exclusión mutua, que haya algo que consumir y que se reinicie algún productor bloqueado, si existe (usando una variable de condición).

```
void producer()
 {
 char x;
 while (true)
 {
 produce(x); //Produce un dato en x
 add(x); //El monitor agrega x al búfer
 }
 }
```

```
void consumer()
 {
 char x;
 while (true)
 {
 x = get(); //El monitor toma un x del búfer
 consume(x); //Consume el dato x
 }
 }
```

```
void main()
   {
      parbegin (producer, consumer);
   }
```

Figura 39. Problema del productor-consumidor usando un monitor.

1.7. Interbloqueo (deadlock)

Los sistemas multiprogramados se basan en la idea de compartir recursos entre varios procesos y por eso son proclives a la ocurrencia de situaciones no deseadas, por ejemplo, las violaciones de la exclusión mutua y la inanición (*starvation*).

Los recursos que solicitan los procesos no siempre están disponibles, lo que provoca esperas que pueden llegar a ser permanentes, por ejemplo, considérese la siguiente situación (figura 40):

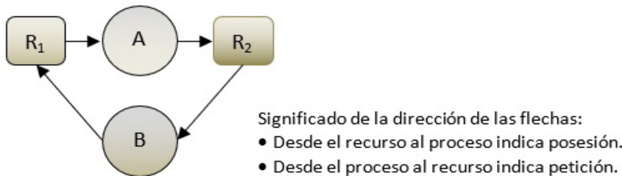


Figura 40. Situación de deadlock entre los procesos A y B.

En un ambiente multiprogramado se están ejecutando dos procesos A y B. El proceso A tiene asignado un recurso único, denominado R1, y B tiene asignado otro recurso único denominado R2 (estos recursos no se pueden compartir), A es dueño del procesador hasta que pide el recurso R2, como el recurso no está disponible A se bloquea (cambio de estado: ejecución-bloqueado) y el SO le da el control al proceso B (cambio de estado: listo-ejecución), el cual ejecuta hasta que necesita el recurso R1 y lo pide, como no está disponible también se bloquea (cambio de estado: ejecución-bloqueado).

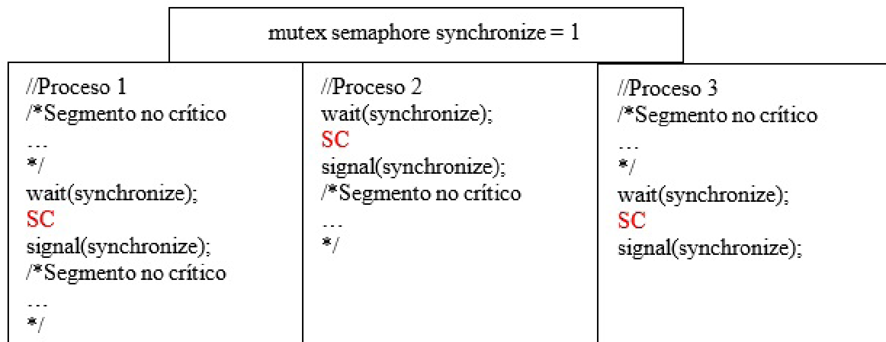
Esta última acción provoca que se produzca una espera que no terminará nunca debido a que A espera por un evento que debe realizar B (liberar a R2) mientras B espera por la liberación de R1 (tiene que realizarla A), como A y B están bloqueándose mutuamente esos eventos no se producirán jamás.

El interbloqueo o deadlock es el bloqueo permanente de un conjunto de procesos provocado por la competencia por los recursos o algún problema de comunicación entre los procesos que están en ese estado indeseado. Esta situación también puede ocurrir en entornos diferentes a los SO.

Un conjunto de procesos está inter bloqueado cuando todos están

bloqueados, en espera de un evento o condición que solo otro proceso del conjunto puede generar (Sol Llaven, 2015).

El deadlock es un problema que ocurre entre procesos concurrentes y a diferencia de los demás problemas de concurrencia no existe una forma eficiente de tratarlo, aunque sí se puede tratar.



mutex semaphore synchronize = 1

//Proceso 1	//Proceso 2	//Proceso 3

/*Segmento no crítico
...
*/
wait(synchronize);
SC
signal(synchronize);
/*Segmento no crítico
...
*/
	wait(synchronize);
	SC
	signal(synchronize);
	/*Segmento no crítico
	...
	*/
		/*Segmento no crítico
		...
		*/
		wait(synchronize);
		SC
		signal(synchronize);

Figura 41. Forma correcta de usar semáforos tipo mutex.

En la figura 41 se utiliza un semáforo tipo mutex, denominando synchronize, para resolver un problema de sincronización entre tres procesos. Para que la solución sea correcta todos los procesos deben ejecutar protocolos a la entrada y la salida de la SC: wait() y signal() respectivamente. Los programadores podrían no seguir la rutina anterior y provocar algunos errores graves, como los que se muestran a continuación:

1. Intercambiar el orden en que se usan las primitivas wait() y signal() o sea hacerlo de la forma siguiente: signal(synchronize); SC; wait(synchronize);

Esta forma de actuar daría la posibilidad a varios procesos de estar dentro de sus SC a la vez, violando la exclusión mutua, lo que traerá resultados impredecibles.

2. Usar wait() en los lugares donde debía estar signal(), o sea: signal(synchronize); SC; signal(synchronize). Este error produce deadlock ya que los procesos se bloquean en espera de que se abra un semáforo que no se abrirá en ningún momento.
3. Por último, si se omite wait(synchronize), signal(synchronize), o ambos podría violarse la exclusión mutua u ocurrir deadlock.

Cuando un sistema cae en deadlock y se logra salir de la situación de alguna manera es difícil reproducir la situación que lo provocó, lo que dificultaría encontrarle una solución (algo típico de los procesos concurrentes).

Condiciones necesarias para que exista deadlock

Para que exista deadlock deben presentarse las cuatro condiciones siguientes, las cuales se conocen como condiciones de Coffman:

- Exclusión mutua. Algunos de los recursos se asignan de manera no compartida.
- Espera con retención. Los procesos que solicitan recursos ocupados retienen los que ya tenían.
- No desalojo. Los recursos asignados a un proceso no pueden expropiarse.
- Espera circular. Debe existir una cadena circular de dos o más procesos, cada uno de los cuales está esperando un recurso retenido por el siguiente miembro de la cadena (Coffman, et al., 1971). Esta última condición es una consecuencia de las anteriores, observe de nuevo la figura 41.

Tratamiento del deadlock

Existen tres estrategias generales para tratar este problema:

1. Prevenirlo. Consiste en hacer fallar una de las cuatro condiciones necesarias.
2. Evitarlo. Antes de asignar un recurso verificar que la asignación es segura, o sea que no puede provocar deadlock más adelante.
3. Detección-recuperación. Establecer algún mecanismo para detectar el deadlock y otro para salir de esa situación.

Prevención del deadlock

Prevenir el deadlock significa violar una de las cuatro condiciones necesarias, lo que debe tomarse en cuenta en el diseño del SO. Se analizarán las condiciones de forma separada.

- Exclusión mutua. Es muy difícil violar esta condición debido a que la naturaleza de muchos recursos obliga a usarlos de forma no compartida; por ejemplo, las impresoras.

- Espera con retención. Puede violarse si se exige que cada proceso pida todos los recursos que necesita cuando comienza, y no los devuelva hasta terminar, de esta manera no esperará por ellos.
- Esta solución es muy ineficiente debido a que los procesos rara vez necesitan todos los recursos simultáneamente y también porque esos recursos permanecerán inactivos por largos períodos de tiempo y los procesos que los necesiten se bloquearán por períodos largos de tiempo.
- No desalojo. Para violar esta condición el proceso que pida un recurso que no esté disponible deberá bloquearse y liberar todos los que poseía (se desalojan). Cuando vuelva a tomar el control tendrá que pedir el recurso que necesitaba y posiblemente los que se vio forzado a liberar.
- Espera circular. Para violarla se establece un orden lineal por tipos de recursos. Si un proceso tiene un recurso de tipo RI solo puede pedir recursos de tipo RJ que sean "mayores" que él; ese orden sigue una lógica de utilización; por ejemplo, se supone que primero se lee y después se escribe, pero es muy difícil establecerlo porque la "lógica" mencionada no tiene que ser cierta, además el método puede resultar extremadamente ineficiente.

Evitar el deadlock.

Los algoritmos para evitar el deadlock utilizan información previa acerca de las necesidades de los procesos y la disponibilidad de los recursos para decidir si satisfacen o no las peticiones recibidas.

Ante cada petición se analizan los recursos disponibles y asignados, así como las futuras peticiones con el objetivo de no caer en algún estado que pueda conducir, en un futuro, a una situación de deadlock.

Existen varios algoritmos para evitar el deadlock, pero solo se analizará uno de ellos.

Algoritmo del banquero. Consideraciones iniciales

Este algoritmo fue concebido por Dijkstra en 1965 y toma el modelo que utilizan los bancos cuando reciben solicitudes de préstamos. El algoritmo se dividirá en dos partes:

- Parte de inicialización.

- Recibe las solicitudes de recursos y simula que los asigna, o sea solo altera las variables controladoras de recursos (VCR) pero no efectúa la asignación.
- Después se invoca el algoritmo de seguridad que analiza si es seguro hacer la asignación (el banco estudia el préstamo): si es seguro se asignan los recursos (el banco hace el préstamo); si no lo es, se restablecen los valores de VCR y se bloquea al proceso solicitante (niega el préstamo).

• La segunda parte corresponde al algoritmo de seguridad. Su trabajo comienza en el estado donde se ha simulado que los recursos fueron entregados.

Tabla 1. Datos para el algoritmo del banquero.

Dato	Significado	Ejemplo
int n	Cantidad de procesos en el sistema	Si n = 3 Existen tres procesos compitiendo por los recursos
int m	Tipos de recursos	Si m = 4 Existen 4 tipos de recursos
int total[m]	Cantidad de recursos de cada tipo m	total[3] = 5 Existen 5 recursos del tipo 3
int demandaM[n, m]	Demanda máxima de cada proceso n por tipo de recurso m	demandaM[1, 3] = 4ç El proceso 1 necesita 4 instancias del recurso tipo 3
int asignados[n, m]	Recursos asignados a cada procesos n por tipo de recurso m	asignados[2, 4] = 1 El proceso 2 tiene asignado un recurso del tipo 4
int pendientes[n, m]	Recursos pendientes por asignar a cada procesos n por tipo m	Necesidad[3, 1] = 2 Al proceso 3 aún le faltan por pedir 2 recursos del tipo 1

La tabla 1, muestra los datos que utiliza el algoritmo, obsérvese que el arreglo total es de una dimensión y los arreglos demandaM, asignados y pendientes son de dos dimensiones. La tabla 2 refleja la situación de un sistema que tiene cinco procesos activos y tres tipos de recursos: 12 del tipo R1, 7 del tipo R2 y 9 del R3.

Tabla 2. Estado de asignación de recursos en un instante dado Sistema con 5 proceso y tres tipos de recursos.

	total			disponibles			demandaM			asignados			pendientes		
Procesos	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_0							9	7	5	0	1	0	9	6	5
P_1							5	4	4	1	0	0	4	4	4
P_2							11	0	4	3	0	2	8	0	2
P_3							4	4	4	3	1	1	1	3	3
P_4							6	5	5	0	0	2	6	5	3
	12	7	9	5	5	4				7	2	5			

Para simplificar la escritura de los algoritmos los arreglos bidimensionales se tratan como vectores, lo que permite sumarlos y restarlos de manera sencilla; por ejemplo, los vectores para el proceso P0 de la tabla 2, son: demandaM0 = (9, 7, 5), asignados0 = (0, 1, 0) y pendientes0 = (9, 6, 5).

El vector pendientesI se obtiene de la manera siguiente: demandaMI - asignadosI, por ejemplo: pendientes0 = demandaM0 - asignados0 = (9, 7, 5) - (0, 1, 0) = (9, 6, 5); por otra parte, disponibles = total - asignados = (12, 7, 9) - (7, 2, 5) = (5, 5, 4).

Seguidamente se presentan dos definiciones que se usarán más adelante.

Secuencia segura

Una secuencia segura de procesos <Pr, Ps,..., Pt>, es aquella en la que cada proceso Pi puede satisfacer sus necesidades pendientes con los recursos que estén disponibles en ese momento más los recursos que están aún ocupados por todos los procesos que le anteceden en la ejecución. Obsérvese que se usan letras como identificadores de los procesos (en este caso: r, s, t, i), porque se desea dejar claro que el subíndice que identifica a los procesos no establece ningún orden con relación a los identificadores de procesos.

Estado seguro

Un sistema está en estado seguro sólo si existe lo que se denomina una

secuencia segura. Una secuencia de procesos < Pv P2, . . ., Pn > es una secuencia segura para el estado de asignación actual si, para cada P¿, las solicitudes de recursos que P, pueda todavía hacer pueden ser satisfechas mediante los recursos actualmente disponibles, junto con los recursos retenidos por todos los P_j, con j < i. En esta situación, si los recursos que P_j necesita no están inmediatamente disponibles, entonces P, puede espe rar hasta que todos los P- hayan terminado. Cuando esto ocurra, P_j puede obtener todos los recursos que necesite, completar las tareas que tenga asignadas, devolver sus recursos asignados y terminar (Silberschatz, et al., 2006).

Debe observarse que un estado no seguro no es un estado de deadlock.

La idea general del algoritmo del banquero es la siguiente. Cada vez que se recibe una solicitud de recursos, se invoca al algoritmo del banquero (figura 42).

```
/*Algoritmo del banquero */
Banquero(peticion_i)
  {
    Si petición_i > demandaM_i
      “Error, el proceso ha excedido la necesidad máxima declarada, salir indicando código del error”
    En caso contrario
      Si petición_i > disponibles
        “No hay suficientes recursos disponibles en este momento, p_i queda en estado de espera”
      En caso contrario
        { /*Alterar los vectores*/
          disponibles    = disponibles    - petición_i    //Quedan petición_i de recursos disponibles
          asignados_i    = asignados_i    + petición_i    //Simula que se asignan petición_i recursos a p_i
          pendientes     = pendientes_i   - petición_i    //Ahora p_i necesitaría petición_i menos de recursos
          Si no (seguridad(disponibles))                  //Estado inseguro, no asignar/
            { /*Restaurar los valores anteriores. Este estado de asignación de recursos no es seguro */
              disponibles = disponibles + petición_i
              asignados_i = asignados_i - petición_i
              necesidad_i = necesidad_i + petición_i
              /*proceso p_i se bloquea en espera de los recursos que necesita */
            }
          En caso contrario                           //Estado seguro
            /*Se asignan los recursos */
        }
  }
```

Figura 42. Algoritmo del banquero.

El cual:

1. Utiliza la petición recibida (peticioni) para alterar los vectores anteriormente analizados y después invocar al algoritmo de seguridad.

2. El algoritmo de seguridad (figura 42) busca una secuencia segura y devuelve éxito o fracaso:

En caso de éxito se entregan los recursos solicitados

En caso de fracaso se deshacen las asignaciones hechas en el paso 1.

Si cada vez que se pida un recurso se invoca al algoritmo del banquero nunca se caerá en deadlock, porque siempre se asignarán los recursos estando en un estado seguro, es decir un estado en el que existe una secuencia segura para salir del caso más extremo (los procesos reclaman todos los recursos que necesitan).

Obsérvese que el algoritmo no exige que se siga esa secuencia (la secuencia real que sigue la ejecución de los procesos es impredecible y dependerá del planificador de periodo corto) sino que se limita a encontrarla cada vez que se piden recursos, garantizando que siempre habrá una salida si se presentará la situación en la que todos los procesos solicitan todos sus recursos pendientes.

```
/*Algoritmo de seguridad */
Seguridad(trabajo) /*El vector trabajo recibe los recursos disponibles */
  {
    Desde i = 1 hasta n                    //Vector finalizado falso. Ningún proceso ha "finalizado" /
      finalizado[i] = Falso
    Desde j = 1 hasta n
      {
        i = 1
        Repetir
          Si (no (finalizado[i]) AND necesidad_i <= trabajo)
            {/* p_i puede finalizar con los recursos disponibles en ese instante, que son los disponibles reales
                más los que liberen los procesos que le anteceden en la secuencia encontrada*/
              finalizado[i] = Verdadero
              /*Cuando el proceso p_i finalice "libera" sus recursos, */
              trabajo = trabajo + asignados_i
              encontrado = Verdadero
            }
          i = i + 1
        Hasta ( i > n) OR ( Encontrado )
      }
    /*Verifica si todos los procesos pueden finalizar de acuerdo a una secuencia segura */
    Desde i = 1 hasta n
      {
        Si no finalizado[i]       //Al menos un proceso no podría finalizar
          Devolver(Falso)         //El estado no es seguro
      }
    Devolver(Verdadero)           //El estado es seguro
  }
```

Figura 43. Algoritmo de seguridad (auxiliar del banquero).

Si cada vez que se pida un recurso se invoca al algoritmo del banquero nunca se caerá en deadlock, porque siempre se asignarán los recursos estando en un estado seguro, es decir un estado en el que existe una secuencia segura que permitiría salir del caso más extremo cuando todos los procesos reclamen todos los recursos que les faltan por pedir (como sucede con los bancos si todos sus clientes solicitan de repente todo el dinero depositado).

La utilización de este tipo de algoritmo puede bajar el rendimiento del sistema ya que en ocasiones puede que se niegue un recurso que está disponible y el proceso solicitante debe quedar en espera para que el estado de asignación de recursos se transforme en un estado seguro.

Detección-recuperación

Las soluciones vistas anteriormente son preventivas, la que se verá seguidamente es no preventiva ya que en este caso se deja que la situación ocurra y cada cierto tiempo se averigua si ha ocurrido.

Cuando se sigue la estrategia de detección es importante establecer cada qué tiempo debe lanzarse el algoritmo, por eso es necesario tomar en cuenta las siguientes interrogantes:

- ¿Con cuánta frecuencia se cree que puede surgir el deadlock?
- ¿Cuántos procesos estarán involucrados en el deadlock cuando ocurra?

Si el deadlock ocurre frecuentemente y la detección demora, bajará el rendimiento del sistema considerablemente, debido a que los recursos que estén asignados a los procesos involucrados estarán inactivos todo ese tiempo.

Mientras más se demore en detectar la situación de deadlock, más se agravará la situación porque aumentará la probabilidad de que más procesos se incorporen a ese estado indeseado.

El deadlock solo puede ocurrir por alguna secuencia de peticiones de recursos no satisfechas que ocasionan el bloqueo de los procesos solicitantes, hasta que llega el último para cerrar la espera circular.

De acuerdo a la observación anterior se pudiera invocar el algoritmo de detección inmediatamente después que se asigne cualquier recurso, de esa forma el deadlock se pudiera detectar en el momento de su

desenlace final.

El problema de la estrategia anterior es que la asignación de recursos ocurre constantemente y el algoritmo de detección es un proceso más que ocupa tiempo, por eso la solución provoca bajo rendimiento en el sistema.

Otra solución sería invocar el algoritmo con una cierta periodicidad estudiada o cuando se detecte que el rendimiento del SO cae, lo que puede ser un indicio de que algo anda mal.

Al igual que en el caso de la evitación del deadlock, los algoritmos de detección necesitan información adicional para poder realizar su tarea.

La figura 44 presenta un algoritmo para detectar el deadlock, obsérvese su similitud con el algoritmo del banquero y la utilización de estructuras de datos similares que son tratadas como vectores.

```
/*Algoritmo para detectar deadlock */
Detectar(trabajo) /*El vector trabajo recibe los recursos disponibles */
  {
    Desde i = 0 hasta n-1
      {    // Solo pueden caer en deadlock los procesos que tienen recursos
           Si asignados_i <> 0   // Si P_i tiene recursos, podría participar
              finalizado[i] = Falso
           En caso contrario    // P_i no puede participar en el deadlock
              finalizado[i] = Verdadero
      }
    Desde j = 0 hasta n-1          // Determina si P_i no está en deadlock
      {
        i = 0
        Repetir
           Si (no (finalizado[i]) AND necesidad_i <= trabajo)
              {
                trabajo = trabajo + asignados_i
                finalizado[i] = Verdadero             // Hasta el momento P_i no está en deadlock
              }
           i = i + 1
        Hasta ( i > n-1)
      }
    /* Verifica si hay deadlock determinando los procesos involucrados*/
    Desde i = 0 hasta n-1
      {
        Si no(finalizado[i])      // P_i está involucrado en el deadlock
           deadlock = Verdadero
      }
    Si no(deadlock)
      devolver(Falso)                            // No existe deadlock
  }
```

Figura 44. Algoritmo para detectar deadlock.

El algoritmo de la figura 44 comienza explorando la asignación de recursos a los procesos. Si un proceso Pi no tiene recursos no puede participar en el deadlock y por eso el índice i del vector finalizado recibe el valor booleano Verdadero, en caso contrario recibe el valor Falso.

Una vez que el algoritmo ha descartado a los procesos que no tienen recursos asignados, procede a explorar los restantes procesos haciendo la verificación necesidadi <= trabajo a todos los procesos que tienen su índice i dentro del vector finalizado con el valor booleano Falso: no(finalizado[i]) AND necesidadi <= trabajo. En este caso se asume que el proceso Pi terminará su tarea con los recursos que tiene asignados y por tanto los devolverá cuando finalice, si esa suposición es falsa podría ocurrir un deadlock que será detectado la próxima vez que se invoque el algoritmo.

1.8. Operaciones sobre procesos

Existen dos maneras generales para trabajar con las facilidades ofrecidas por los SO:

- La más sencilla es a través de un conjunto de programas, conocidos comúnmente como comandos o utilitarios, que se invocan desde una terminal (de texto o gráfica) o se incluyen dentro de algún programa, casi siempre de tipo script.
- La otra manera es a través de programas de más alto nivel; por ejemplo, C, desde los cuales se invocan servicios ofrecidas por el núcleo del SO a través de llamadas al sistema.

Los comandos pueden ser internos o externos, los primeros forman parte del intérprete de comandos (Shell) y los segundos son programas que se distribuyen junto con el SO. Los comandos (externos o internos) no forma parte del SO, al igual que el Shell, aunque se distribuyen junto con él.

Comandos de los SO tipo Unix

Los comandos externos de los SO tipo Unix residen en los directorios /bin o /bin/usr. La tabla 3 muestra algunos que son útiles para el trabajo con procesos. En el anexo 1 pueden observarse otros comandos generales, aunque no está entre los objetivos del libro hacer un análisis detallado de ellos.

Tabla 3. Algunos comandos UNIX para el trabajo con procesos.

Comando	Uso	Ejemplos	
		Ejemplo	Resultado
at	Ejecuta comandos a la hora especificada	at -f file 10:00	Ejecuta los comandos contenidos en el archivo file a las 10 am
gcore	Crea una imagen del proceso	gcore 1437	Crea una imagen del proceso con pid 1437 que puede usarse para encontrar errores
ipcs	Imprime datos acerca la comunicación entre procesos	ipcs -m	Brinda un reporte sobre los segmentos de memoria compartida
kill	Fuerza la terminación de procesos	kill 1534	Termina el proceso 1534
nice	Ejecuta un comando o programa con la menor prioridad, o sea es “gentil” con los demás	nice program	Ejecuta program asignándole la menor prioridad posible
nohup	Continúa la ejecución de un comando después que el usuario que ordenó ejecutarlo salió del sistema	nohup program	Continúa ejecutando program después que el usuario hizo logout
ps	Reporta los procesos activos	ps -u mlezcano	Ofrece un listado de los procesos que se ejecutan a nombre del usuario mlezcano
sleep	Espera los segundos especificados para iniciar la ejecución de un comando	ls; sleep 20; ps	Imprime un listado (ls) del directorio actual y espera 20 segundos para imprimir los procesos activos (ps)
wait	Espera que todos los procesos que se ejecutan de fondo terminen o espera la terminación de un proceso especifico	wait 2045	Espera la terminación del proceso 2045

El programa script que se presenta en la figura 45 está hecho en el lenguaje Bash (interprete de comandos muy popular en los SO tipo Unix). Obsérvense las aclaraciones que se hacen a continuación:

```
#!/bin/bash
echo "-----Reporte generado-----"                                    > Reporte
echo "El tipo de SO en uso es $OSTYPE"                                >> Reporte
echo "El Shell por defecto es: $SHELL"                                >> Reporte
users=`who | wc -l`
echo "En este momento hay $users usuarios utilizando el sistema"     >> Reporte
process=`ps -U "$USER" | wc -l`
echo "El usuario actual es $USER."                                    >> Reporte
echo "En su nombre se ejecutan los siguientes $process procesos:"    >> Reporte
ps -U "$USER"                                                         >> Reporte
```

Figura 45. Ejemplo de programa en bash script.

- La primera línea del programa, #!/bin/bash, le ordena al intérprete de comandos en uso que cargue y ejecute el programa bash que está localizado en el directorio /bin. De ahí en lo adelante el bash será responsable de ejecutar las órdenes contenidas en el archivo.

Si no se incluye esta línea dentro del archivo, las órdenes serán ejecutadas por el intérprete de comandos en uso, lo cual podría arrojar algunos errores si las órdenes están destinadas a otro shell.

- Se usan los símbolos > y >> para redirigir las salidas:
  - El símbolo > hará que la salida estándar (el monitor) se redirija al archivo Reporte, sobrescribiéndolo si existe.
  - El símbolo >> también redirige la salida estándar, pero en este caso los contenidos se agregan al final del archivo Reporte.
  - En ambos casos, si el archivo no existe se crea.
- El comando interno echo, se usa para mostrar una línea de texto.
- En el programa se usan dos tipos de variables:
  - De entorno o ambiente: OSTYPE, contiene el tipo de SO; SHELL, contiene la ruta absoluta hacia el intérprete de comandos y USER, contiene el login del usuario.
  - De usuario: users y process.
- La orden users=`who | wc -l` asigna a la variable de usuario users el resultado de evaluar el comando compuesto who | wc -l. Los apóstrofos invertidos encerrando una expresión, a la derecha de una sentencia de asignación (=), hacen que se ejecute toda la expresión.

El comando who lista los usuarios que están activos en el sistema, su salida se envía a través de una tubería (|) hacia el comando wc -l que

cuenta las líneas del reporte emitido por who y por tanto la variable users recibe el resultado final: la cantidad de usuarios activos.

- La orden process=`ps -U "$USER" | wc -l` tiene una idea similar a la explicada anteriormente, pero en este caso ps -U "$USER" hace que se listen todos los procesos que se ejecutan a nombre del usuario (Lezcano Brito, 2018) contenido en la variable USER.

No se incluye entre los objetivos de este texto el estudio de ningún lenguaje de programación, el lector interesado en mayores detalles acerca del lenguaje Bash script puede remitirse a los capítulos IV y V del libro "Fundamentos de sistemas operativos", un estudio más detallado se ofrece en el texto "Advanced Bash-Scripting Guide".

1.8.1. Uso de llamadas al sistema en los SO tipo Unix

La figura 12, mostró un ejemplo del uso de las llamadas al sistema fork() y wait()de los SO tipo Unix, seguidamente se formalizan esas llamadas al sistema.

Las llamadas al sistema wait(), waitpid() y waitid() esperan por el cambio de estado de un proceso hijo. Los cambios de estados ocurren cuando:

- El hijo termina.
- El hijo se detiene debido a una señal recibida.
- El hijo se reinicia por una señal recibida.

El SO libera los recursos asociados a los procesos que terminan esperas por llamadas al sistema tipo wait provocadas por la terminación de esos procesos. Si un hijo termina sin haber sido esperado por el padre, se convierte en un proceso "zombie".

El *kernel* mantiene un conjunto mínimo de información sobre los procesos zombies lo que permite a sus padres efectuar operaciones wait() posteriores para obtener datos acerca de la terminación de los hijos que habían finalizado sin esperar por sus padres. Si un padre termina, sus proceso zombies son adoptados por el proceso init el cual se crea cuando se inicia el SO y es el padre de todos los procesos en los SO tipo Unix, y permitirá efectuar una operación wait() para eliminarlos.

Todos los procesos, incluidos los zombies, están contenidos dentro de una tabla de procesos que es finita; si esa tabla se llena no se podrán crear más procesos y en ese caso la llamada al sistema fork() fallará.

Las llamadas al sistema wait(), waitpid() y waitid() provocan una espera al proceso que la invoca pero tienen algunas particularidades que se distinguen a continuación. La figura 46 muestra la sintaxis de estas tres llamadas al sistema.

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
int waitid(idtype_t idtype, id_t id, siginfo_t *infop, int options);
```

Figura 46. Sintaxis de las llamadas al sistema wait(), waitpid() y waitid().

Llamada al sistema wait()

La llamada al sistema wait() el proceso padre invocará dicha llamada al sistema, la cual hará que quede en espera hasta que él proceso hijo termine (Silberschatz, et al., 2006).

Llamada al sistema waitpid()

La llamada al sistema waitpid() suspende la ejecución del proceso que la invoque hasta que el hijo especificado por el argumento pid cambie de estado. Los valores de pid pueden ser:

- < -1 Espera por cualquier proceso hijo con identificador de grupo (gid) igual al valor absoluto de pid.
- -1 Espera por cualquier hijo.
- 0 Espera por cualquier hijo que tenga identificador de grupo igual a la del proceso que invoca la llamada al sistema.
- > 0 Espera por el proceso hijo especificado.

Por defecto esta llamada solo espera por la terminación del hijo pero ese comportamiento puede modificarse a través del argumento options que se forma con una operación OR sobre las constates siguientes:

- WNOHANG Retorna inmediatamente si el hijo no ha salido.
- WUNTRACED Retorna si un hijo se ha detenido.
- WCONTINUED Retorna si el hijo detenido ha sido reiniciado por la entrega de la señal SIGCONT.

Si la variable status no es NULL, wait() y waitpid() almacenan información en el entero al que apuntan. Ese valor puede inspeccionarse usando la variable status como argumento de las siguientes macros:

- WIFEXITED(status). Devuelve true si el hijo termina normalmente, o sea invocando la función exit() o a la llamada al sistema _exit(); o cuando retorna

desde la función main().

- WEXITSTATUS(status). Devuelve el estado de salida del hijo que se refleja en los 8 bits menos significativos del argumento status que el hijo especificó en exit(), _exit() o como argumento de la sentencia return de main(). Solo puede utilizarse si WIFEXITED retorna true.
- WIFSIGNALED(status). Devuelve true si el proceso hijo termina por una señal.
- WTERMSIG(status). Devuelve el número de la señal que causó el fin de la espera. Solo puede utilizarse si WIFSIGNALED retorna true.
- WCOREDUMP(status). Devuelve true si el proceso hijo produce un vaciado del núcleo. Solo puede usarse si WIFSIGNALED retorna true. No está disponible en algunas implementaciones UNIX.
- WIFSTOPPED(status). Devuelve true si el proceso hijo fue detenido por la entrega de una señal. Solo es posible si la llamada se hizo usando WUNTRACED o cuando se le está siguiendo una traza al hijo.
- WSTOPSIG(status). Devuelve el número de la señal que provocó la detención del hijo. Solo se puede emplear si WIFSTOPPED devuelve true.
- WIFCONTINUED(status). Devuelve true si el proceso hijo se ha reiniciado por la entrega de la señal SIGCONT.

Llamada al sistema waitid()

La llamada al sistema waitid() ofrece un control más preciso sobre los cambios de estado de los procesos por los que se espera.

Los argumentos idtype e id seleccionan los hijos por los que se espera

de acuerdo a las siguientes especificaciones:

- Si idtype == P_PID espera por el hijo especificado en id.
- Si idtype == P_PGID espera por cualquier hijo con GID igual al valor de id.
- Si idtype == P_ALL espera por cualquier hijo y se ignora el valor de id.

Los cambios de estados por los que debe esperarse se especifican a través de una operación OR con las banderas siguientes:

- WEXITED espera por la terminación del hijo.
- WSTOPPED espera por un hijo detenido por la entrega de una señal.
- WCONTINUED espera porque se reinicie un hijo, previamente detenido, debido a la entrega de la señal
- WNOHANG Se comporta como waitpid().
- WNOWAIT Deja al hijo en un estado de espera que permitirá una llamada wait() posterior para obtener información de su estado.

Si waitid() tiene éxito fija los campos siguientes de la estructura siginfo_t que está apuntada por infop:

- si_pid contiene el id del proceso hijo.
- si_uid contiene el id real del proceso hijo. Algunas implementaciones UNIX no fijan este campo.
- si_signo siempre se fija a SIGCHLD.
- si_status depende del estado de salida del hijo dado por _exit() o exit(), o de la señal que provocó su terminación, detención o reanudación. El campo si_code puede usarse para interpretarlo y contendrá uno de los valores siguientes:
  - o LD_EXITED el hijo llamó a _exit().
  - o CLD_KILLED el hijo fue eliminado por una señal.
  - o CLD_DUMPED el hijo fue eliminado por una señal y se hizo un vaciado del núcleo.
  - o CLD_TRAPPED fue atrapada una traza sobre el hijo.
  - o CLD_CONTINUED el hijo continuó al recibir la señal SIGCONT.

Valores de retorno de la familia de llamadas al sistema wait

Cuando fracasan devuelven -1 y si tienen éxito:

- wait() devuelve el identificador (pid) del hijo que terminó.
- waitpid() devuelve el identificador del hijo (pid) que cambió de estado.
- waitid() devuelve 0.

Llamada al sistema fork()

La llamada al sistema fork() (figura 47), crea un proceso (denominado proceso hijo) que es un duplicado del proceso que lo crea (nombrado proceso padre), excepto en que el hijo:

- Tiene su propio y único identificador de proceso.
- No hereda los bloqueos de memoria de su padre, los ajustes de semáforos, los registros de bloqueo ni los temporizadores.
- Las estadísticas de utilización de recursos y contador de uso de la CPU se inician en cero.
- El conjunto de señales pendientes se inicia vacío.

```
#include <unistd.h>

pid_t fork(void);
```

Figura 47. Sintaxis de la llamada al sistema fork().

Llamadas al sistema getpid(), getppid()

```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

Figura 48. Sintaxis de las llamadas al sistema getpid() y getppid().

La llamada al sistema getpid() retorna el pid del proceso que la ejecuta, mientras getppid() devuelve el pid de su proceso padre (figura 48).

Llamadas al sistema execve()

La llamada al sistema execve(), permite ejecutar un programa (se convertirá en proceso). La figura 49 muestra su sintaxis, el programa a ejecutar está apuntado por la variable filename, la variable argv apunta a los argumentos pasados al programa que se ejecutará y la variable

envp adopta la forma convencional key=value, donde key representa una variable de ambiente y value es el valor que se le asigna a dicha variable.

Llamadas al sistema pause()

La llamada al sistema pause() provoca la espera por una señal (figura 50)

```
#include <unistd.h>

int execve(const char *filename, char *const argv[], chat *const envp);
```

Figura 50. Sintaxis de las llamada al sistema execve().

```
#include <unistd.h>

int pause();
```

Figura 51. Sintaxis de las llamada al sistema pause().

A continuación, se presentan dos ejemplos del uso de las llamadas al sistema analizadas anteriormente. El primer ejemplo está formado por dos programas que se usan en forma combinada (figuras 50 y 51).

El programa de la figura 52, denominado exec1.c., sigue los pasos siguientes:

1. Comienza inicializando las variables apuntadoras a arreglos: newargv[] y newenviron[].
2. Después comprueba que se le pase un argumento: argc != 2. Si esta comprobación falla, sale del programa con código de error 1: exit(1). La variable argc contiene la cantidad de argumentos pasados al programa más uno.
3. Si la comprobación anterior tiene éxito, invoca la llamada al sistema fork() para crear un proceso hijo, asignándole a la variable pid la salida de fork(): pid = fork().
4. El nuevo proceso (el hijo) ejecuta la llamada al sistema exeve() que cargará el código binario obtenido a partir del programa exec2.c (figura 52), mientras el proceso padre se queda esperando a que el hijo termine.

```
//Programa exec1.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc, char *argv[])
   {
      char *newargv[]          = { "primero", "segundo", "tercero", NULL                             };
      char *newenviron[]       = { "PATH=/usr/bin", "MyVariable=/home/mlezcano", NULL                };
      pid_t pid;

      if (argc != 2)
         {
         printf("La sintaxis correcta es: %s <archivo a ejecutar> \n", argv[0]);
         exit(1);                  //Salida con error. Código 1
         }
      pid = fork();                //Se crea un nuevo proceso
      if(pid == 0)                 //Código del proceso hijo
         {
            execve(argv[1], newargv, newenviron);            //Ejecuta el código apuntado por argv[1]
            printf("Error en la ejecución de execve:\n");    /* execve() no retorna, por eso solo se llega aquí si hay
                                                              error en su ejecución */
            exit(2);               //Código de error 2
         }
      else  //Código del proceso hijo
         {
            printf("Soy el proceso padre %d, hijo del shell %d, esperaré por mi hijo %d\n", getpid(), getppid(), pid);
            wait();                //El proceso padre espera la terminación de su único hijo
            printf("Soy el proceso padre %d, terminó la espera por mi hijo %d\n", getpid(), pid);
         }
      exit(0);                     //El padre termina sin error, código 0
   }
```

Figura 52. Uso de llamadas al sistema: programa exec1.c.

```
//Programa exec2.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *env[])
   {
      int j;

      printf("Soy el proceso %d hijo de %d ejecutando la llamada al sistema execve()\n", getpid(), getppid());
      printf("Argumentos que recibí por parámetros:\n");
      for (j = 0; j < argc; j++)
         printf("argv[%d]=%s\n", j, argv[j]);
      printf("Variables de ambiente que recibí por parámetros:\n");
      for (j = 0; env[j] != NULL; j++)
         printf("env[%d]=%s\n", j, env[j]);
      printf("Variables de ambiente generales:\n");
      system("env");
      printf("Terminará el proceso hijo %d\n", getpid());
      exit(0);
   }
```

Figura 53. Uso de llamadas al sistema: programa exec2.c.

Para ejecutar estos programas deberán seguirse los pasos siguientes:

1. Compilar los programas desde una terminal UNIX (se usa gcc):

Para obtener el código binario e1: gcc -o e1 exec1.c

Para obtener el código binario e2: gcc -o e2 exec2.c

2. Ejecutar el programa binario e1, pasándole como argumento el programa binario e2: ./e1 e2

La figura 54 presenta un programa que utiliza algunas de las facilidades de las llamadas al sistema tipo wait.

```
//Programa exec2.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[], char *env[])
  {
    int j;

    printf("Soy el proceso %d hijo de %d ejecutando la llamada al sistema execve()\n", getpid(), getppid());
    printf("Argumentos que recibí por parámetros:\n");
    for (j = 0; j < argc; j++)
      printf("argv[%d]=%s\n", j, argv[j]);
    printf("Variables de ambiente que recibí por parámetros:\n");
    for (j = 0; env[j] != NULL; j++)
      printf("env[%d]=%s\n", j, env[j]);
    printf("Variables de ambiente generales:\n");
    system("env");
    printf("Terminará el proceso hijo %d\n", getpid());
    exit(0);
  }
```

Figura 54. Ejemplo de llamadas al sistema: programa wait.c.

Aunque no tiene mayores complejidades es conveniente hacer algunas aclaraciones:

- El programa comienza intentando crear un proceso hijo (cpid = fork();), si no lo logra emite un mensaje de error (perror("…")) y sale señalándolo (exit()).
- Si fork() tiene éxito, crea un nuevo proceso que ejecuta el código enmarcado dentro de los comentarios: /* Código del hijo */ -y- /* Fin del código del hijo */, donde se quedará en espera de alguna señal (pause() provoca la espera).
- El padre ejecuta el código enmarcado dentro de los comentarios: /* Código del padre */ -y- /* Fin del código del padre */ y usa waitpid()

para esperar por el hijo con identificador cpid. La variable status, que se utiliza como segundo parámetro de waitpid(), almacenará información acerca de la terminación del hijo, mientras el tercer parámetro es una operación OR entre las constantes WUNTRACED (retorna si el hijo se ha detenido) y WCONTINUED (retorna si el hijo detenido se reinicia por la señal SIGCONT).

En el capítulo se han estudiado algunos conceptos claves para analizar los SO, quizás el más importante de ellos es el de proceso, que se define en su forma más simple, como un programa en ejecución; o sea es una entidad activa que realiza varias tareas, a diferencia de los programas que son entidades pasivas que no hacen nada.

Los procesos necesitan recursos para realizar su actividad, esos recursos pueden ser de *hardware* (memoria, procesador, entre otros.) o de *software* (archivos abiertos, tablas de directorios, entre otros).

Para planificar el uso de los procesadores centrales (puede ser uno o más) existen diferentes algoritmos, lo cuales pueden usar el desalojo o no. Los algoritmos con desalojo le retiran el procesador a los procesos, aunque no hayan terminado su ciclo de ejecución. Los algoritmos sin desalojo solo le retiran el procesador a los procesos cuando estos necesitan realizar ciertas operaciones; por ejemplo, de E/S, que son satisfechas por los periféricos.

Los procesos se identifican, típicamente, por un número entero y están representados dentro del SO por una estructura de datos conocida como Bloque de Control de Proceso (PCB), que es su CPU virtual, debido a que en ella se guardan los valores que necesita el proceso para restaurarse después de haber sido interrumpido.

Los procesos pueden ser pesados o ligeros (hilos) y su vida transcurre por diversos estados que describen su actividad, por ejemplo: listo, ejecutando, terminado, bloqueado, entre otros.

Cuando los procesos ejecutan concurrentemente debe prestarse atención especial a los recursos que se usan de forma exclusiva. El mal uso de estos recursos puede provocar condiciones de competencia con resultados impredecibles, es por eso que en las partes de código donde se violen las condiciones de Bernstein, denominadas secciones críticas, debe seguirse algún protocolo para adquirir y liberar los recursos de

manera apropiada.

El deadlock es uno de los diversos problemas que existen cuando se comparten recursos, para tratarlo se usan tres estrategias: prevenirlo, evitarlo y detectarlo.

1.9. Ejercicios

1. **Explique las diferencias entre planificación con desalojo y sin desalojo.**
2. **¿Cuáles son las dos etapas fundamentales que sigue un proceso cuando se está ejecutando? Explíquelas.**
3. **Cuando se trabaja en sistemas multiprogramados es importante lograr una buena mezcla de trabajo ¿Qué es una buena mezcla de trabajo y por qué es importante?**
4. **Mencione y explique los algoritmos de planificación que se han analizado en este texto.**

a) ¿Cuál cree que sea "el mejor"? Justifique su respuesta con argumentos sólidos.

b) Busque en la bibliografía otros algoritmos de planificación y analícelos.

5. **Con relación al PCB.**

a) Explique su importancia.

b) ¿Cuándo se utiliza el PCB?

c) ¿Por qué se dice que el PCB es la CPU virtual de cada proceso?

6. **Con relación a la operación dual.**

a) Explique cómo funciona.

b) Explique los problemas a los que se enfrenta un SO si el procesador para el que se hace solo tiene un modo de procesamiento.

7. **La sección crítica es un segmento de código que viola las condiciones de Bernstein.**

a) Explique las tres condiciones de Bernstein.

b) Explique, por separado, las condiciones que debe satisfacer una solución al problema de la sección crítica.

c) ¿Qué significa que una solución tenga espera ocupada? ¿Por qué no es recomendable la espera ocupada?

8. **Los SO pueden usar tres tipos de planificadores.**

a) Mencione y explique cada uno de ellos.

b) Explique cómo actúan entre sí.

9. **¿Cuál es la relación que existe entre el planificador de periodo corto y el despachador?**

10. **Ofrezca una definición para proceso pesado y otra para proceso ligero en la que queden claras las diferencias entre ellos.**

11. **Diga los momentos en que un proceso puede cambiar de estado y explique por qué se produce ese cambio.**

12. **Defina el concepto de estado seguro.**

13. **Explique por qué un estado seguro no puede conducir al deadlock.**

14. **Verifique si el estado que se describe en la tabla 2 es un estado seguro.**

15. **Diga si un estado inseguro conduce siempre a un estado de deadlock. Explique.**

16. **Dibuje el árbol de procesos que se corresponde con la ejecución de:**

a) Del programa de la figura 49 (observe que está asociado al programa de la figura 50).

b) Del programa de la figura 51.

Capítulo II. Administración de la memoria

2.1. La memoria y su estructura

Desde el punto de vista físico se le denomina memoria a cualquier dispositivo capaz de almacenar datos. La memoria se estructura en varios niveles jerárquicos:

- Registros: memoria de alta velocidad y poca capacidad que forma parte de la arquitectura del procesador. Permite guardar valores temporalmente para hacer operaciones sobre ellos, ocupa el nivel superior de la jerarquía.
- Memoria caché: memoria más rápida y cara que la memoria interna, pero con igual funcionalidad. Tiene el propósito de agilizar los accesos del procesador a la memoria interna. La capacidad de los registros, generalmente, se mide en bits (8, 16, 32, 64).
- Memoria primaria, interna o principal: está conectada a la CPU por buses (de direcciones, de datos y de control), es volátil (igual que la caché) y de mayor costo que las que le siguen en la jerarquía.

Se utiliza para almacenar los procesos (programas en ejecución) y sus datos. Este capítulo se dedica al manejo de la memoria interna.

Es usual referirse a ella sin adjetivo, acompañando a los demás tipos de memoria de los adjetivos adecuados, esa convención se usará en este texto y por tanto en lo adelante la palabra memoria hará referencia a la memoria interna.

- Memoria externa o secundaria. Proporciona capacidad de almacenamiento por largos períodos de tiempo y es mucho mayor y más lenta que la memoria interna.

En el capítulo I se analizaron diversas formas para planificar la utilización del procesador central; el objetivo principal de los algoritmos que se mostraron es explotar ese componente de *hardware* de manera eficiente. Para lograr la meta trazada es necesario distribuir el tiempo del procesador entre diferentes procesos; los cuales tienen que cargarse en memoria, parcial o totalmente, con el propósito de ejecutarse.

La memoria puede verse como un arreglo lineal que permite almacenar un byte o una palabra (cadena finita de bits que la computadora maneja como un conjunto; hoy en día las palabras son de 16, 32 o 64 bits,

pero han existido otros tamaños) en cada una de sus localizaciones, cada localización puede accederse por su dirección.

La figura 55 muestra una memoria interna de 5 bytes, su primera localización (con dirección 0) contiene la letra D, y la última (con dirección 4) contiene la letra S. Es muy poco probable que exista un dispositivo con tan poca memoria y solo se toma como ejemplo.

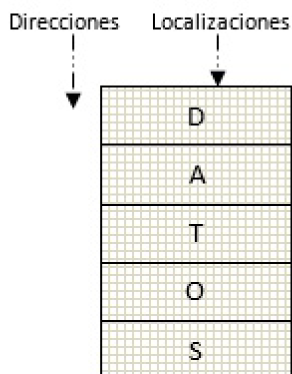


Figura 55. Memoria de 5 bytes.

La capacidad de la memoria ha crecido muchos en los últimos tiempos, pero siempre resulta insuficiente; aun cuando se trabaja en ambientes multiprogramados donde es necesario repartirla entre todos los procesos que se estén ejecutando.

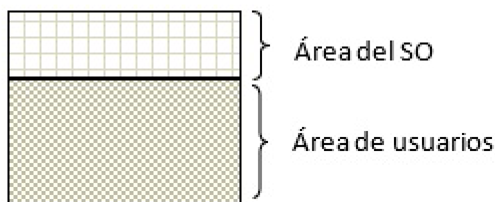


Figura 56. Dos zonas de memoria, una para el SO y otra para los usuarios

Normalmente la memoria puede verse dividida en dos zonas, una para el núcleo o *kernel* del SO y otra, denominado área de usuarios, destinada a cargar los procesos y sus datos (figura 56). En los sistemas multiprogramados la zona de usuario se reparte entre todos los procesos que han sido admitidos en el sistema, mientras en los monoprogramados el único proceso de usuario que existe se adueña de toda esa parte de

la memoria.

Con el objetivo de tener el procesador ocupado la mayor cantidad de tiempo posible, es necesario que los SO multiprogramados tengan suficientes procesos cargados en memoria, de esa manera cuando algunos de ellos se bloqueen por E/S se le puede asignar el procesador a cualquier proceso que esté listo.

Cada proceso debe alojarse en una zona única de memoria, protegida de accesos no permitidos, aunque debe existir alguna estrategia para compartir algunas zonas cuando sea necesario. La protección de la memoria será responsabilidad del procesador debido a que es imposible que el SO pueda predecir las localizaciones que ocupará un programa, mientras es responsabilidad del SO dividir la memoria en distintas partes que serán ocupadas por diferentes procesos.

2.2. Particiones fijas

La estrategia de particiones fijas no se usa actualmente, pero es un buen punto de partida para comprender mucho de los problemas que deberá tratar el SO.

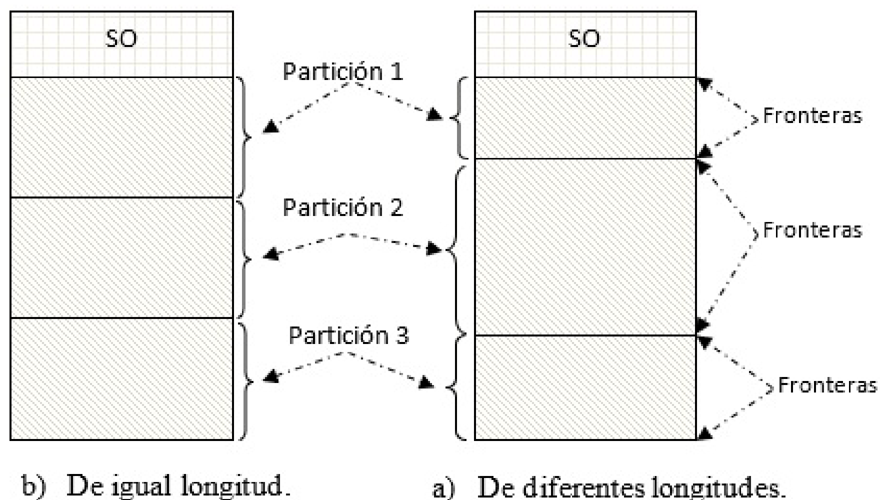


Figura 57. Particiones fijas.

La idea se basa en dividir la memoria del área de usuarios en varias

partes, denominadas particiones (figura 57), de manera que se pueda cargar un proceso en cada una de ellas. Las particiones pueden ser de igual o de diferentes longitudes; parte izquierda y derecha de la figura respectivamente.

En este caso la protección de las diferentes particiones se garantiza por la existencia de dos direcciones que establecen los límites (superior e inferior) de cada partición. Esas direcciones, denominadas fronteras, pueden guardarse en registros especiales del procesador central.

Las dos principales dificultades de esta estrategia de asignación de memoria son:

- Si la longitud del programa es mayor que la de cualquier partición, no cabrá en memoria y el programador tendrá que lidiar con ese problema.
- La memoria se utiliza ineficientemente debido a que cada programa se adueña de la partición que se le asigne, aun cuando su necesidad sea menor que la capacidad de memoria brindada por la partición, lo que deja un espacio sin usar que puede ser considerable. Este problema se conoce como fragmentación interna y no tiene solución debido a que la partición completa queda reservada para el proceso que la ocupa, o sea está bloqueada para los demás procesos.

Cuando todas las particiones son de igual tamaño, la asignación de memoria es muy fácil ya que no es relevante el hecho de asignar una partición u otra a un proceso dado. En esta estrategia hay una cola de procesos listos que compiten por cualquier partición (figura 58).

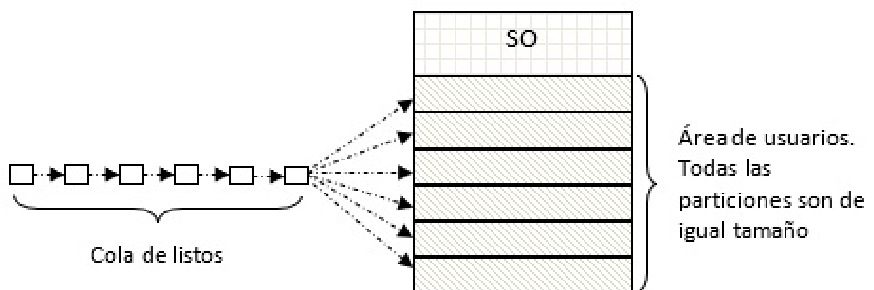


Figura 58. Particiones fijas de igual longitud.

Cuando las particiones tienen diferentes longitudes, la asignación de memoria debe seguir una estrategia de asignación más selectiva que

puede ser:

- Tener una cola por cada partición y asignar los procesos, de manera permanente, a una de las colas que deberá estar asociada a la menor partición que satisfaga las necesidades de cada proceso, lo que provocaría la menor fragmentación interna posible.

Esta técnica parece tener todo a favor, sin embargo puede provocar que algunos procesos estén detenidos a pesar de que existen particiones libres mayores que su necesidad donde se podrían alojar.

- Tener una sola cola de listos y tratar de asignar la partición más pequeña posible, pero sin dejar de usar las particiones libres aun y cuando parezcan muy grandes.

Con esta idea se mantiene la posibilidad de disminuir la fragmentación interna, pero no se deja de ejecutar un proceso cuando hay particiones libres que satisfacen sus necesidades, lo que mejora el rendimiento del sistema en general. Queda claro que no podrá satisfacerse la petición de memoria hecha por algún proceso con necesidad de memoria mayor que cualquier partición libre y en ese caso el proceso solicitante tendrá que esperar a que se libere una partición que tenga una longitud mayor o igual que sus necesidades.

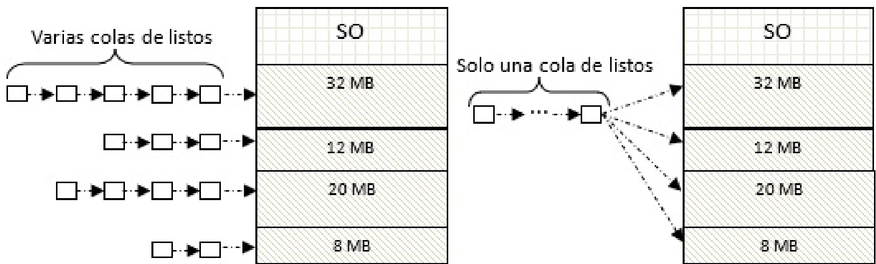


Figura 59. Particiones fijas de diferentes longitudes.

La figura 59 muestra las dos maneras de tratar las particiones fijas, una implementación de esta idea se materializó en el SO OS/MFT (Multiprogramming with a Fixed number of Task) para una máquina *mainframe* IBM 360.

Una situación especial se presenta cuando todas las particiones están ocupadas y se desea ejecutar un proceso distinto a los que están en

memoria. En ese caso el SO puede desalojar un proceso en favor de otro.

La figura 60 muestra un ejemplo en el que se desaloja el proceso P2 para cargar el proceso P3 en su lugar, al que se le dará el control del procesador.

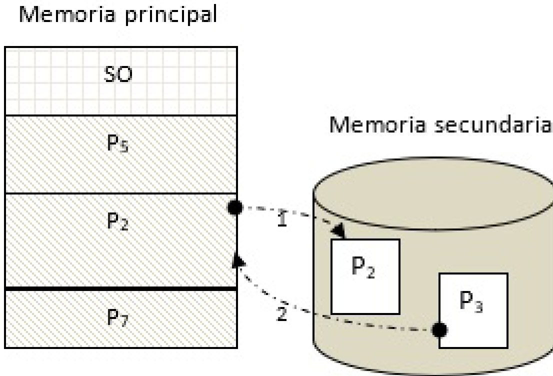


Figura 60. Particiones fijas. Intercambio de procesos.
(1) Desalojo de P2 y entrada de P3.

2.3. Particiones variables

Las particiones variables surgieron para tratar de resolver los problemas de las particiones fijas y tampoco se usa actualmente, el SO OS/MVT (Multiprogramming with a Variable Number of Tasks) hecho para máquinas *mainframe* IBM utilizó esta técnica (figura 61).

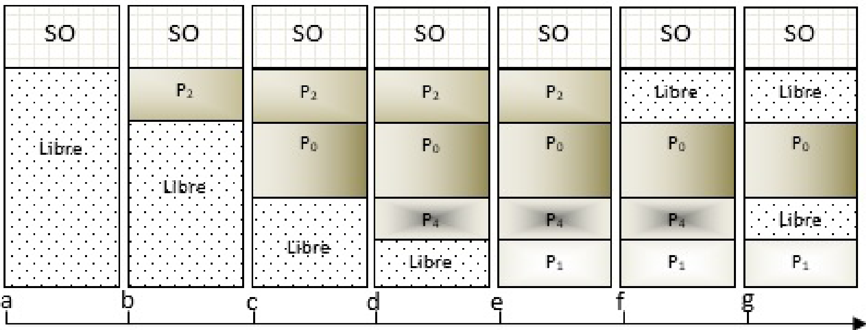


Figura 61. Particiones variables. Instantes a, b, c, d, e y f .

En este caso el SO ve la memoria de usuario como un gran espacio libre que se irá dividiendo en particiones de diferentes tamaños, de acuerdo

a las solicitudes recibidas. A cada proceso se le asigna la cantidad exacta de memoria que necesita y por eso no hay fragmentación interna. En la idea esquematizada de la figura 61, se puede apreciar que:

- En el instante a, solo está el SO en memoria y por eso toda la memoria de usuario está libre.
- En el instante b se carga el proceso P2 que ocupa los primeros espacios libres dejando un "hueco" a partir de su última localización.
- En los instantes c, d, y e, entran los procesos P0, P4 y P1 respectivamente, quedando toda la memoria ocupada.
- Posteriormente, instante f, el proceso P2 termina y por último en el instante g termina P4.

Después de esta secuencia de asignaciones y liberaciones de espacios, la memoria queda fragmentada (figura 61 instante g). Si llegara un proceso que tuviera una longitud mayor que cualquiera de las dos particiones libres pero menor que la suma de esos dos espacios, no se podría satisfacer. Este problema se conoce como fragmentación externa.

La fragmentación externa puede resolverse si se desfragmenta la memoria para lo cual será necesario mover los procesos, con el objetivo de dejar el espacio libre contiguo. Para el caso de la figura 62 podría moverse el proceso P0 hacia arriba para dejar el "hueco" en el medio, tratando de mover la menor cantidad de procesos posibles. La desfragmentación es costosa y compleja porque será necesario relocalizar las direcciones de los procesos movidos, lo cual también puede consumir un tiempo apreciable. El resultado de desfragmentar la figura 61 se observa en la figura 62.

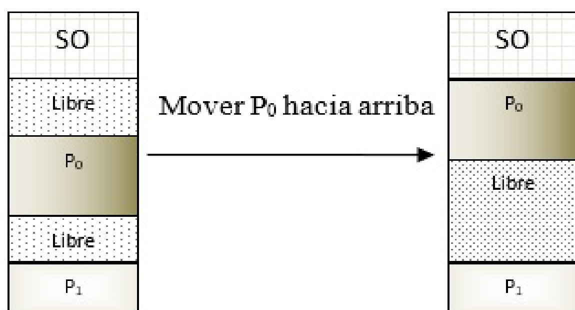


Figura 62. Desfragmentación. Solo es necesario mover el proceso P0.

La política de particiones variables debe usar algún algoritmo de asignación para elegir la partición que se usará cuando exista más de una disponible, se consideran tres estrategias que se denominan: primer acceso, mejor acceso y peor acceso.

En general para satisfacer una petición de memoria de tamaño m, debe encontrarse un espacio de tamaño n tal que $m \leq n$, a continuación, se ofrecen detalles de cada estrategia.

- El primer acceso. Consiste en tomar el primer espacio que satisfaga la condición.
- El mejor acceso. Busca el espacio de menor tamaño que satisfaga la petición, para lo cual deberá explorarlos todos.
- El peor acceso también explora todos los espacios disponibles, pero en este caso escoge el mayor de todos.

Es fácil observar que la estrategia del mejor acceso puede dejar pequeños espacios libres que posiblemente no sean adecuados para ningún proceso, provocando fragmentación externa; por otra parte, la estrategia del peor acceso deja espacios libres mayores (Lezcano Brito, 2018) que pueden ser mejores candidatos para utilizarse; por último, no es posible predecir lo que sucede con el primer acceso.

Como se ha visto, las dos políticas analizadas tienen algunos inconvenientes, la de particiones fijas limita la cantidad de procesos activos y por tanto el grado de multiprogramación y también es ineficiente con relación al uso del espacio; por otra parte, la de particiones variables es más compleja, necesita de la desfragmentación y por todo eso es más difícil de mantener.

En cualquier caso, debe tomarse en cuenta el problema de la relocalización. Si se usan particiones fijas, con procesos asignados permanentemente a colas específicas por cada partición, solo es necesario calcular las direcciones cuando los procesos entren a memoria por primera vez, para lo cual se le sumará a cualquier referencia a una dirección relativa la dirección base de la partición donde se alojará el proceso.

Para que el lector comprenda el problema de la relocalización se representa la estructura interna de un proceso conformado por: su identificador, una zona para código, una para datos y otra para la pila.

La parte derecha de la figura 63, muestra el código en C, pero en

realidad será un código binario ya traducido. El punto de entrada del proceso (su primera instrucción ejecutable), es la sentencia pid = fork(); que está desplazada d bytes desde la dirección 0 del proceso. La llamada a la función make() provoca un salto (1) desde la dirección actual hacia make(), de la cual se regresa (2) cuando se ejecute la sentencia return. La pila se usa, entre otras cosas, para guardar esas direcciones y poder realizar los saltos mencionados.

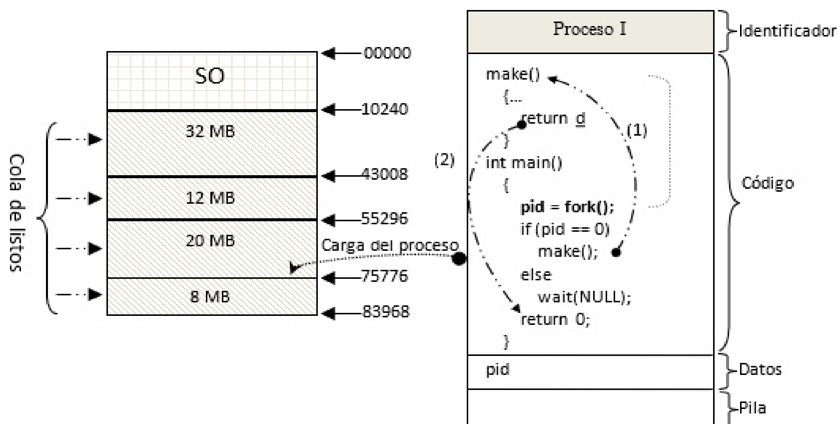


Figura 63. Proceso I, se carga en la última partición de memoria.

En la figura 63, el proceso se carga en la última partición que comienza en la dirección 75776, de manera que la primera instrucción ejecutable se calcula como: 75776 + d. También será necesario calcular la dirección donde realmente queda la función make(), en la figura está en la posición 0 del proceso de ahí que su dirección al cargarse en la última partición será 75776.

Las direcciones de los procesos no cargados en memoria van desde 0 hasta su longitud y se denominan direcciones lógicas, las direcciones generadas al calcular el lugar real que ocupan los procesos en memoria se nombran direcciones físicas. El mapeo de direcciones lógicas sobre direcciones físicas es una de las labores más importantes del módulo de administración de la memoria de cualquier SO. El cálculo de las direcciones se hace con el apoyo del *hardware*.

Cuando se usan particiones fijas de igual longitud y una sola cola y cuando se utilizan particiones variables, el problema se torna más com-

plejo debido a que los procesos que se desalojen de memoria, no tendrán ninguna garantía de alojarse en la misma partición que estaban cuando se carguen nuevamente, de manera que en esos momentos será necesario relocalizar las direcciones físicas. Para el caso de las particiones variables también es necesario relocalizar las direcciones de los procesos después de haberlos movidos para desfragmentar la memoria.

El SO usa el valor de las fronteras entre las particiones para detectar cualquier error de direccionamiento, o sea los procesos solo pueden referirse a direcciones que estén entre sus fronteras (inferior y superior) y cualquier violación será detectada.

2.4. Paginado

Tanto las particiones fijas como las variables exigen que los procesos se carguen completos y de manera contigua en memoria. En esta sección y en la siguiente se analizan dos técnicas de manejo de memoria que eliminan esas restricciones, lo que permite manipular la memoria más eficientemente.

El paginado se basa en la idea de dividir la memoria física en unidades de igual longitud, denominadas marcos de páginas o frames, mientras que la memoria lógica de cada proceso se divide en unidades de ese mismo tamaño nombradas páginas. Cada proceso posee una tabla de página que se usa para mapear sus direcciones lógicas en direcciones físicas y existe una lista global de marcos de páginas libres. La figura 64 muestra la idea esquematizada.

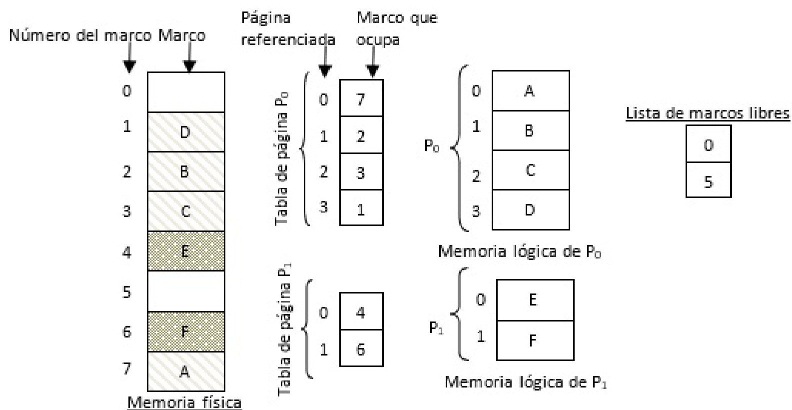


Figura 64. Esquema paginado.

En la figura 64 se han representado dos procesos, con sus respectivas tablas de páginas. La memoria física tiene ocho marcos de páginas (de la 0 a la 8), mientras la memoria lógica del proceso P0 tiene 4 páginas y la del proceso P1 posee 2 páginas.

La tabla de página de P0 indica que sus páginas están cargadas en los marcos de páginas: 7, 2, 3 y 1. La tabla de página de P1 muestra que sus páginas se localizan en los marcos de páginas: 4 y 6. En ese momento hay dos frames libres (0 y 5).

Debe observarse que las páginas de un proceso no tienen que estar contiguas en la memoria física (como sucede con las particiones fijas y variables). Para distinguir las páginas de los procesos P0 y P1, en la figura 56, se han colocado letras a sus páginas, así como una trama distinta en los marcos de páginas ocupados por cada uno de ellos.

Debido a que las páginas y los marcos de páginas tienen igual longitud, la tabla de página solo necesita indicar en qué marco de página está cargada una página dada. De esta manera, para conocer la dirección de inicio de una página deberá multiplicarse su longitud (lp) por el marco en que está localizada; por ejemplo, si se supone que las páginas de la figura 64 son de 1024 bytes, las direcciones de inicio de las páginas del proceso P1 son:

- Para la página 0 que está en el marco de página 4: 4 * 1024 = 4096
- Para la página 1 que está en el marco de página 6: 6 * 1024 = 6144

Para encontrar la dirección real o física (df) que ocupa una dirección lógica (dl) cualquiera dentro de la memoria física debe conocerse:

1. La página (p) a la que pertenece la dirección lógica, lo que se determina por una simple operación de división entera: p = dl div lp.
2. El desplazamiento (d) que hay a partir de la dirección 0 de la página hasta la posición que ocupa la dirección lógica, lo que se calcula encontrando el resto de la división entera: d = dl mod lp.

Seguidamente se analiza un ejemplo que se basa en la figura 64. Supóngase que durante la ejecución del proceso P1 se ha encontrado una referencia a la dirección lógica 1098 (se asumirá que es un salto a esa posición). En ese momento es necesario determinar la posición exacta en que está cargada la instrucción referida, por eso debe conocerse

la página, del proceso P1, a la que pertenece la dirección, así como el desplazamiento que tiene dentro de esa página. El procedimiento de cálculo es el siguiente:

1. Calcular la página a la que pertenece la dirección lógica 1098 del proceso P1: 1098 div 1024 = 1.
2. Calcular el desplazamiento de la dirección lógica 1098 dentro de la página 1 del proceso P1: 1098 mod 1024 = 74.

O sea la dirección lógica 1098 está formada por el par (1, 74), donde el primer elemento es la página a la que pertenece esa dirección y el segundo es su desplazamiento dentro de esa página.

Una vez calculado el par (página, desplazamiento) se busca, en la tabla de página del proceso P1, el marco de página en que está cargada la página 1, en este caso es el 6. La dirección de inicio del marco de página 6 se obtiene por una simple operación de multiplicación: 6 * 1024 = 6144 y esa es la dirección de inicio de la página 1 dentro de la memoria física, a la que habrá que sumarle el desplazamiento (74) para obtener la dirección física de la dirección lógica 1098 (6144 + 74 = 6218). Observe la figura 65.

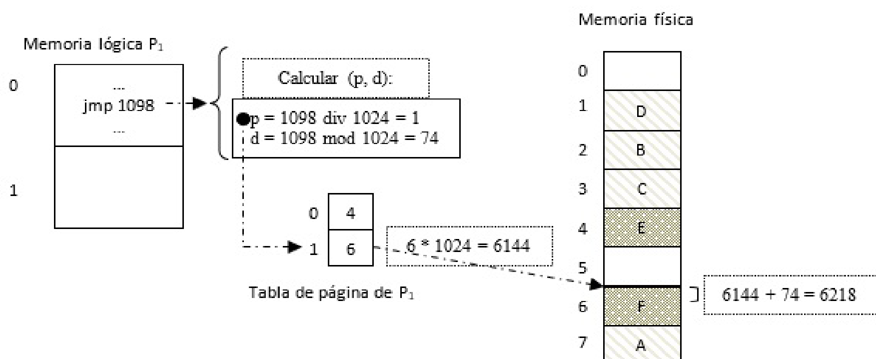


Figura 65. Cálculo de la dirección física 1098 del proceso P1.

El mapeo de direcciones físicas en direcciones lógicas lo realiza el *hardware*, mientras es responsabilidad del SO mantener las tablas de páginas y la lista de frames libres. Para realizar ese mapeo se fija la restricción de que la longitud de las páginas, y por tanto de los marcos de páginas, sea potencia de 2, con lo cual se simplifican las operaciones

explicadas anteriormente.

La figura 66, es equivalente a la 65, pero en la primera se usa notación decimal y en la segunda notación binaria. Se han supuesto direcciones de 16 bits; obsérvese que no es necesario sumar, basta con concatenar el marco de página con el desplazamiento.

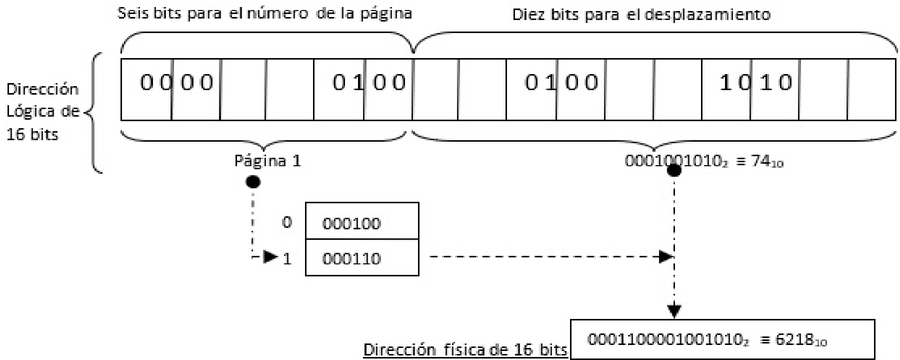


Figura 66. Versión de la figura 2.11 en formato binario.

Cuando se usan longitudes de páginas que son potencia de 2 las mismas direcciones contienen la página y el desplazamiento de la dirección, por ejemplo, para páginas de 1024 bytes (210), se necesitan 10 bits para acceder al mayor desplazamiento dentro de una página: 102310, equivalente a 1 1 1 1 1 1 1 1 1 1 2, lo que deja 6 bits para el número de la página (las direcciones de páginas comienzan en 0).

Lo mismo sucedería si las direcciones fueran de 32 o de 64 bits, donde de nuevo los últimos 10 bits representan el desplazamiento dentro de la página (para páginas de 1024 bytes) y lo bits más significativos, o sea los primeros (22 y 54 respectivamente), representan el número de la página. Queda claro que el tamaño de la página fija la longitud del campo desplazamiento.

El paginado no sufre fragmentación externa debido a que los marcos de páginas se pueden asignar a cualquier proceso sin necesidad de que sus páginas estén contiguas. Además, debido a que las páginas y los marcos de páginas tienen la misma longitud (Lezcano Brito, 2018), estos últimos siempre satisfacen cualquier solicitud.

Por otra parte, el paginado puede sufrir fragmentación interna en su

última página, la cual no siempre se usa totalmente, aunque sí se asigna completa. Por ejemplo, si se tiene un trabajo con un tamaño de 8200 bytes y se usan páginas de 2048 bytes, se puede conocer cuántos marcos de páginas necesita el proceso para alojarse completo en memoria realizando la operación 8200 / 2048, de la que se obtiene cociente 4 con resto 8, es decir el proceso necesita 4 páginas más 8 bytes, pero solo pueden asignarse páginas completas y por eso el sistema le dará 5 páginas al proceso. Debe observarse que solo se usan 8 bytes de la última página y por eso quedan 2040 bytes de fragmentación interna.

2.5. Segmentado

Desde el punto de vista de los programadores la división de la memoria lógica de los procesos en páginas no resulta natural. Esta afirmación se debe a que, por lo regular, el programador se inclina a pensar en un programa como un conjunto de unidades lógicas que se forman de acuerdo a algún objetivo, por ejemplo: un procedimiento, un método o función para realizar una tarea específica; la pila, un conjunto de datos asociado a una finalidad dada, entre otros.

Estas unidades lógicas tienen diferentes tamaños y poseen una funcionalidad que, casi siempre, está totalmente definida dentro de ella misma, por ese motivo cada unidad se trata como un todo denominado segmento. Observe la figura 67.

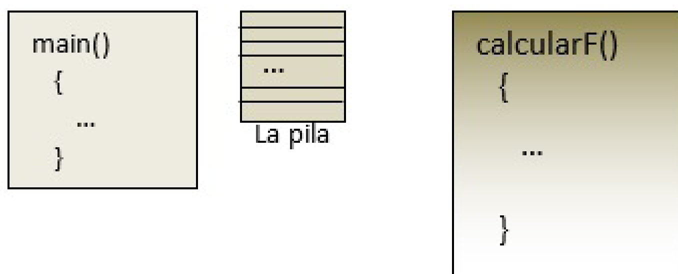


Figura 67. Tres segmentos de distintos tamaños.

En esta concepción, al igual que en el paginado, las direcciones lógicas están formadas por un par (segmento, desplazamiento), su primer componente es el número del segmento y el segundo el desplazamiento de la dirección dentro del segmento (a partir de su dirección 0). Los segmentos, al igual que las páginas, no tienen que estar de forma con-

tigua, de ahí que un proceso puede estar cargado en varios segmentos dispersos por la memoria.

No existe fragmentación interna en el segmentado porque a cada segmento se le asigna la memoria exacta que necesita, pero la liberación de espacios asignados a segmentos separados entre sí provoca fragmentación externa, al igual que en la estrategia de particiones variables analizada en el epígrafe 2.2, aunque en el segmentado es menor debido a que los segmentos tienden a ser más pequeños.

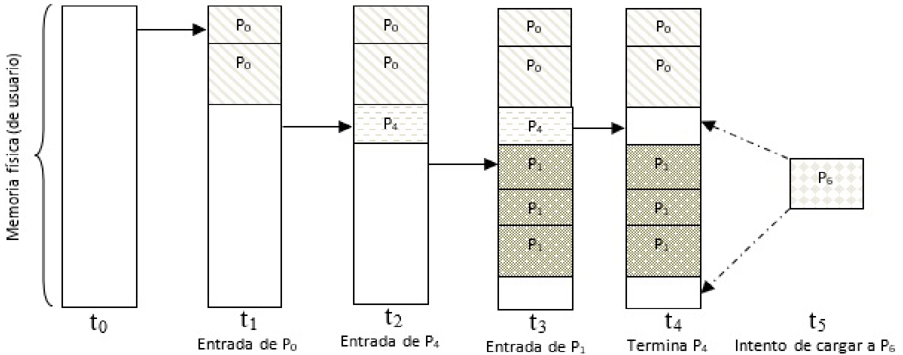


Figura 68. Asignación dinámica de memoria segmentada y fragmentación externa.

Observe la figura 68, al inicio toda la memoria de usuario está libre (instante t0), en el instante t1 entran en memoria dos segmentos del proceso P0, en t2 entra un segmento de P4 y en t3 entran tres segmentos de P1. A cada proceso se le asigna la memoria en forma segmentada, de acuerdo a la longitud de cada uno de sus segmentos, lo que va dejando cada vez menos espacio.

En el instante t4 termina el proceso P4 que libera su espacio; en ese momento han quedado dos "huecos" separados. En el instante t5 llega la solicitud de memoria para el proceso P6, su único segmento es mayor que cualquiera de los dos espacios no contiguos que quedan en memoria, pero menor que la suma de esos dos espacios, provocándose la fragmentación externa.

Para el programador la forma en que se maneja la memoria paginada es transparente, pero por lo regular no es así en el segmentado. Esta realidad puede complejizar un poco la programación, pero permite organi-

zar los programas y sus datos de la manera que resulte más conveniente.

Mientras en el paginado se utilizan tablas de páginas (una por cada proceso) para encontrar las localizaciones físicas de las direcciones lógicas, en el segmentado se utilizan tablas de segmentos, pero en este caso no existe una relación simple entre las direcciones lógicas y las físicas, debido a las diferencias de longitudes entre los segmentos. Las tablas de segmentos y la lista de segmentos libres, tienen que especificar la dirección de entrada de cada segmento o bloque libre y su longitud.

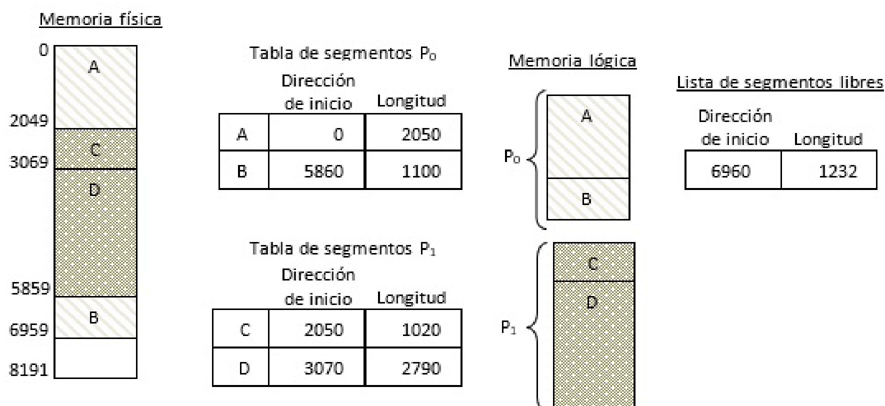


Figura 69. Ejemplo de uso de memoria segmentada.

En la figura 69 se tiene una memoria física de 8192 MB para usuarios, que está ocupada por dos procesos (P0 y P1), cada uno con dos segmentos (la cantidad de segmentos no tienen que ser igual). En ese instante solo queda un segmento libre, obsérvese la necesidad de que las tablas de segmentos y la lista de segmentos libres tengan al menos dos campos: uno para las direcciones de inicio y otro para las longitudes. También debe observar que el proceso P0 no está contiguo en memoria.

Resulta más conveniente analizar el esquema segmentado en notación binaria. En ese caso, cada dirección lógica puede verse como un número que tiene dos partes:

- · Los n bits más significativos representan el número del segmento (Lezcano Brito, 2018).

- • Los m bits menos significativos representan el desplazamiento de la dirección lógica dentro de ese segmento, de manera que la longitud máxima de un segmento es 2m.

El *hardware* sigue el siguiente algoritmo para mapear direcciones lógicas en direcciones físicas:

1. Extraer los n bits más a la izquierda de la dirección lógica, obteniendo un número que es un índice a la tabla de segmentos (TS), sea i.

Extraer los m bits más a la derecha, ellos indican el desplazamiento; asignarlo a D.

2. Extraer:

TS[i].Dirección de inicio, asignarlo a DI.

TS[i].Long, asignarlo a Long.

3. Si Long >= D el desplazamiento es mayor que la longitud del segmento y por eso no es una dirección válida. Informar error y salir.
4. Dirección física = DI + D. Retornar.

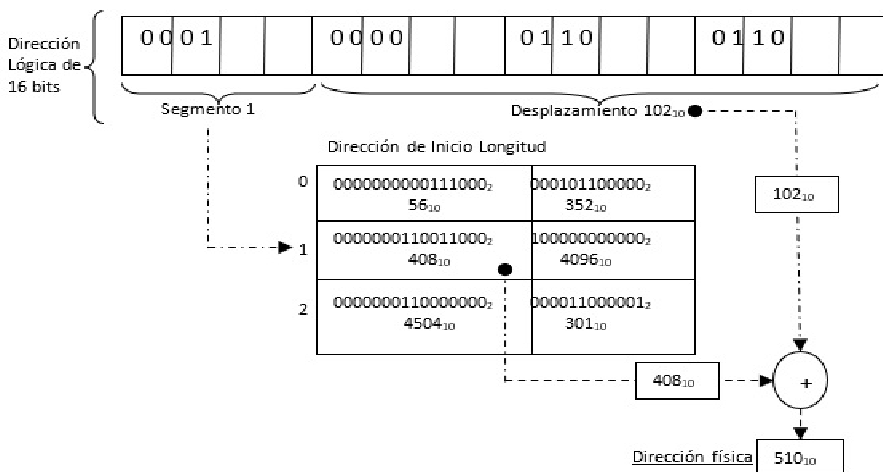


Figura 70. Segmentado: cálculo de una dirección física.

El ejemplo de la figura 70 se basa en direcciones de 16 bits que tienen los primeros 4 bits dedicados al número del segmento y los restantes 12 bits para el desplazamiento. En ese caso la longitud máxima de un seg-

mento es 212 (4096 bytes). Se han situado los correspondientes valores en notación decimal para que se comprenda mejor la idea.

Cuando un proceso pasa al estado de ejecución, el *hardware* carga la dirección de su tabla de segmentos en un registro especial (Lezcano Brito, 2018) del procesador que se utiliza para manejar la memoria.

2.6. Memoria virtual

La palabra virtual se usa para describir algo que se comporta como real a pesar de no serlo, en particular la memoria virtual hace referencia a una extensión de la memoria interna sobre memoria externa.

En realidad, el paginado y el segmentado siempre están ligados a la memoria virtual, es decir, a la concepción analizada en los epígrafes 2.4 y 2.5 debe agregársele el manejo de la memoria virtual que es la manera real en que se utilizan esas técnicas.

El uso de la memoria virtual se basa en el hecho de que los procesos no ejecutan todas sus sentencias a la vez, de manera que no es necesario cargarlos completos en memoria para comenzar a ejecutarlos, es decir pueden cargarse las partes (en este contexto una parte puede ser una página, un segmento o cualquier otra unidad en que se divida la memoria) que los componen de acuerdo a las sentencias que se vayan a ejecutar. Es una técnica de gestión de la memoria que permite que el sistema operativo disponga, tanto para el *software* de usuario como para sí mismo, de mayor cantidad de memoria que esté disponible físicamente (García Chango & Salazar Ramón, 2018).

Los esquemas paginados y segmentados se prestan, de manera natural, para apoyar la idea esbozada en el párrafo anterior, debido a que en ellos el código se divide en partes (páginas o segmentos) que pueden cargarse cuando son demandadas, adicionalmente cuando la memoria está llena pueden desalojarse algunas partes para poder cargar otras.

Esta última observación hace necesaria la existencia de algún equipo externo, generalmente un disco, para resguardar las partes de los procesos que no han entrado en memoria o que salen de ella, posiblemente de manera temporal.

Tanto en el paginado como en el segmentado la traducción de direcciones lógicas en direcciones físicas ocurre en tiempo de ejecución, si

no fuera así no sería posible que las partes que componen los procesos entren y salgan de la memoria sin preocuparse del lugar real en que se ubican en cada momento. Los fragmentos de procesos que permiten calcular sus direcciones reales en tiempo de ejecución se denominan relocalizables.

Una ventaja del uso de la memoria virtual es que el programador no tiene que ocuparse por el tamaño de sus programas ya que ellos se cargarán por partes, o sea es posible ejecutar programas que son mayores que la memoria física. En cualquier caso, siempre es necesario que el código que se vaya a ejecutar esté en memoria interna, de ahí que a la memoria física, en este ámbito, se le denomine memoria real mientras al espacio de todas las direcciones de un proceso se le nombre memoria virtual o lógica.

Para que un proceso cualquiera se pueda ejecutar es necesario que esté en memoria, sin importar el esquema de manejo utilizado, pero si se usa memoria virtual no es necesario que esté completo. En este caso el SO solo carga las partes que necesita para comenzar (código y datos) y cuando se haga referencia a alguna dirección lógica que no esté en memoria, será necesario cargar la otra parte que la contiene, para lo cual se usan las tablas de página, de segmento o cualquier otra estructura de datos que permita realizar esta labor.

El trabajo con la memoria virtual se realiza de manera conjunta entre el SO y el procesador. Cuando este último encuentra una dirección lógica que no está en memoria principal genera la interrupción falta de acceso a memoria, el SO reacciona cambiando el estado del proceso interrumpido de listo a bloqueado, le asigna el procesador a otro proceso y envía una petición de E/S al disco (donde están almacenados todos los procesos).

El manipulador del disco localiza la parte demandada, la carga en la memoria y envía una interrupción de E/S al SO para avisar que terminó su trabajo, en ese momento el SO cambia el estado del proceso interrumpido de bloqueado a listo y lo sitúa en la cola de listos para que compita por el procesador.

Debe observarse que cuando el SO hace la petición de E/S al disco, le asigna el procesador a otro proceso para no desperdiciar el tiempo

que tardará el manipulador de ese equipo en satisfacerse la petición hecha.

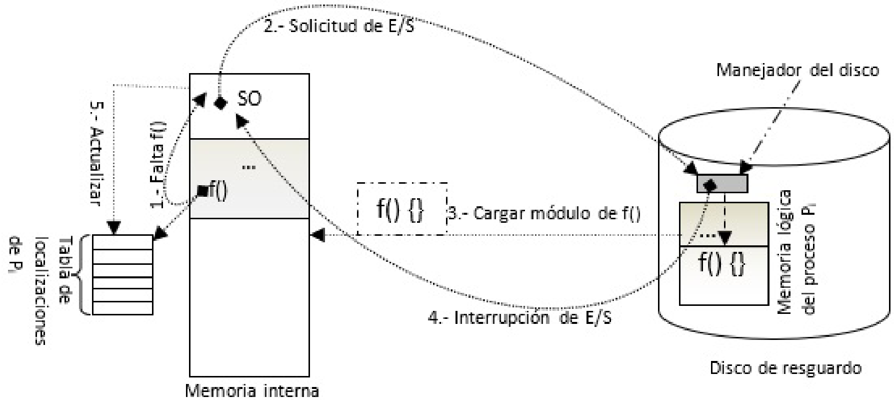


Figura 71. Falta de una "parte" del proceso.

La idea se esboza en la figura 71, obsérvese el algoritmo que se sigue cada vez que una referencia a memoria provoque una falta de acceso (se asume que el proceso Pi está parcialmente cargado en memoria). El proceso Pi ejecuta hasta que encuentra la referencia a la función f(), que no está en memoria; en ese instante se desencadenan los pasos siguientes:

1. El procesador genera una interrupción falta de acceso a memoria que es atendida por el SO (los pasos se corresponden con los números en la figura).

2. El SO cambia el estado del proceso Pi, de listo a bloqueado (en el PCBi) y envía una solicitud de E/S al disco de resguardo o intercambio (swap).

La petición de E/S puede satisfacerse de inmediato o puede ir a una cola de peticiones al disco, en dependencia de las solicitudes previas que existan.

3. El manejador del disco carga la parte solicitada en algún lugar de la memoria.

4. El manejador del disco envía una interrupción de E/S al SO para avisarle que ya cargó la parte solicitada en la memoria.

5. El SO actualiza la tabla de localizaciones del proceso Pi, cambia el estado del proceso de bloqueado a listo y lo pone en la cola de listos para competir por el procesador.

Los dos principales beneficios de esta estrategia son:

- Se pueden tener más procesos en memoria, lo que beneficia la multiprogramación y hace más eficiente la utilización del procesador.
- Los procesos pueden ser mayores que la memoria principal lo que resulta muy importante para los programadores que no tendrán que lidiar con ese problema.

Los beneficios de la memoria virtual no se pueden negar y por eso los SO actuales, en general, usan esta técnica. Por otra parte, tampoco se puede negar que se pierde tiempo del procesador cuando se llevan y se traen partes hacia y desde la memoria física. Cuando el SO gasta más tiempo de procesamiento en llevar y traer partes hacia y desde la memoria que en ejecutar instrucciones de los procesos se produce una condición que se denominará tráfico de partes en este texto y que en inglés se conoce como trashing.

Las investigaciones en el campo de los SO han obtenido diversos algoritmos para evitar el thrashing, muchas de ellas se basan en la idea de adelantarse al futuro cercano, en particular es muy importante el principio de localidad de las referencias que se basa en el hecho de que durante la ejecución de un programa las referencias a memoria hechas por el procesador tienden a agruparse, por ejemplo:

- Muchos programas contienen ciclos iterativos y funciones que referencian a un pequeño conjunto de instrucciones.
- Las operaciones sobre tablas y arreglos acceden a conjuntos de datos agrupados.

Tomando en cuenta estas consideraciones, es posible organizar el código y los datos localmente para que los accesos estén cercas uno de otros y así evitar, en algo, las referencias a localizaciones que estén en partes distintas.

2.6.1. Memoria virtual paginada

En el epígrafe 2.4 se estudió el esquema paginado sin tomar en cuenta la memoria virtual, ahora se retoma el tema para analizarlo desde esta otra perspectiva.

En este caso no es necesario cargar todas las páginas de los procesos y por eso la tabla de página debe incluir nuevos campos para poder determinar si las páginas están o no en memoria.

Un bit, denominado bit de presencia (bit P) especifica si una página está en memoria; por ejemplo, si contiene un cero (0) la página referida no está en memoria y si su valor es uno (1) sí lo está.

Un segundo bit, nombrado bit de modificación o bit M identifica las páginas que han sido modificadas durante el tiempo que han permanecido en memoria. Si el bit M de una página dada tiene valor 1 la página se modificó y si tiene valor 0 no ha sido modificada durante el tiempo que permaneció en memoria, en este último caso no será necesario actualizar su copia en disco (en su memoria lógica), lo que ahorra tiempo.

De acuerdo al SO, al algoritmo de reemplazamiento utilizado (se verá más adelante) y a otras necesidades pueden existir otros campos en la tabla de página.

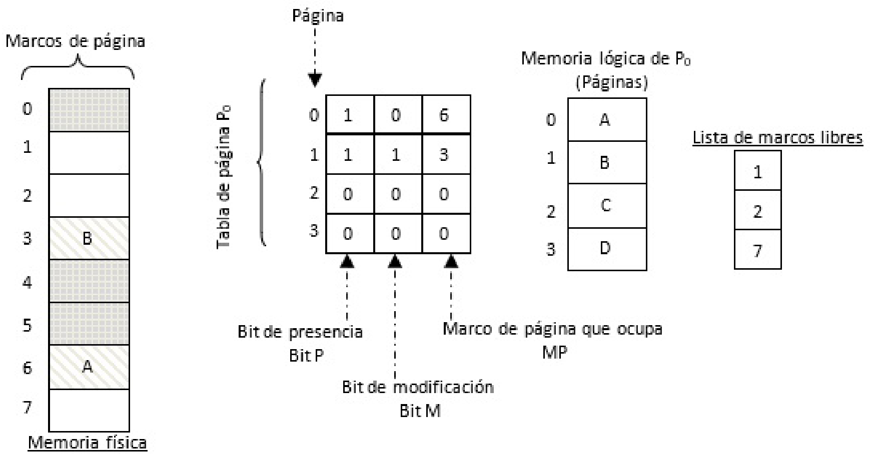


Figura 72. Paginado con memoria virtual.

La figura 72 muestra un ejemplo de lo analizado anteriormente. En este caso el proceso P0, tiene cuatro páginas: solo las páginas 0 y 1 están en memoria en ese instante (bit P con valor 1), la página 1 ha sido modificada (bit M con valor 1) y la página 0 no ha sido alterada (bit M con valor 0). Las páginas 2 y 3 no están en memoria (bit P con valor 0).

Las tablas de páginas son de longitud variable y por eso no es posible cargarlas en los registros del procesador, pero sí es posible mantenerlas en memoria principal para agilizar los accesos y hacer que un registro apunte a su dirección de inicio.

Tabla 4. Longitudes de páginas.

Procesador	Longitud de de la página
DEC Alpha	8 KB
UltraSPARC	de 8 KB a 4 MB
Pentium	4 KB o 4 MB
Intel Itanium	de 4 KB a 256 Mbytes
Intel core i7	De 4 KB a 1 GB

Como la tabla de página puede ser grande, algunos SO también la paginan; es decir solo mantienen una parte de ella en memoria principal y el resto en memoria virtual.

Una consideración importante de diseño de *hardware* es determinar la longitud de las páginas, la tabla 4 muestra algunos ejemplos. Existen varios factores que deben tomarse en cuenta para tomar la decisión correcta:

- Uno de ellos es la fragmentación interna que se analizó previamente. Queda claro que si se escogen páginas pequeñas se reduce este problema, pero los procesos necesitarán más páginas y por tanto las tablas de páginas serán mayores, lo que pudiera traer como consecuencia que algunas de sus partes estén en memoria virtual, en este caso habría una falta de página doble (una para traer la parte de la tabla de página que falta y otra para traer la página en sí) haciendo más lentos los accesos.
- En segundo lugar, la mayoría de los equipos actuales de almacenamiento masivo (externo o secundario) se apoyan en un mecanismo rotacional. Para este tipo de equipo es preferible que las páginas sean grandes debido a que la transferencia de datos hacia y desde la memoria se hace por bloques.

La figura 73 muestra la situación de un proceso Pi que tiene cuatro páginas (observe su memoria lógica en el disco). La tabla de página (TP)

muestra que dos de las páginas están cargadas en memoria, la 0 y la 1 (bit P = 1); la página 0 no se ha modificado (bit M = 0) mientras la página 1 ha sufrido cambios durante su estancia en la memoria (bit M = 1), el último campo de la TP, denominado Mp, indica los marcos de páginas en que están cargadas las páginas. También existe una lista de marcos libres (LML). En esta figura hay páginas de otros procesos cargados en memoria, pero no interesan para este análisis.

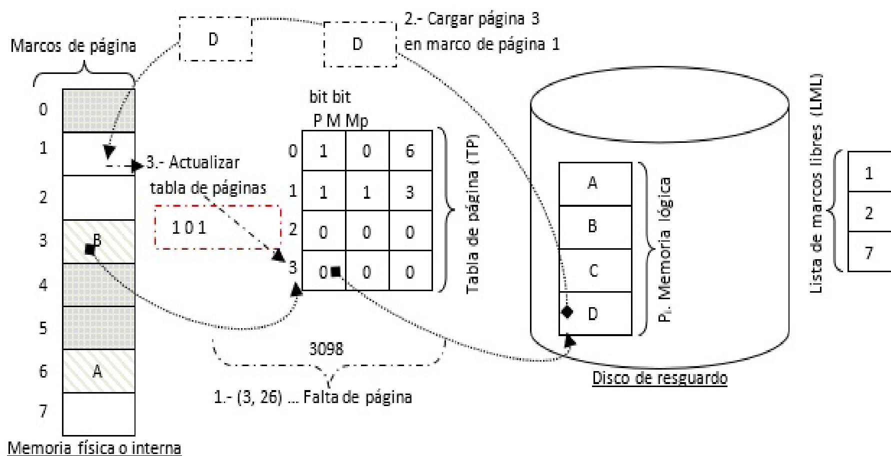


Figura 73. Falta de página. Referencia a pagina 3 desplazamiento 26.

Supóngase que las páginas de la figura 65 son de 1024 bytes, por eso el proceso Pi necesitaría 1024*4 bytes para cargarse totalmente en memoria, en ese instante solo se han cargado dos páginas. En un instante dado desde la página 1 (señalada con una B) se hace referencia a la dirección lógica 3098, el cálculo 3098/1024 da por resultado 3 con resto 26, o sea el par página desplazamiento es (3, 26), entonces para mapear la dirección lógica 3098 en su dirección física o real será necesario buscar, en la tabla de páginas de Pi, la localización de la página 3, pero su bit P está en 0, o sea la página 3 no está en memoria (se produce una falta de página).

La falta de página anterior provoca la búsqueda de un marco libre para cargar la página solicitada. Se escoge, de LML, el marco 1 y se carga la página 3 de Pi en el marco de página 1. Después se actualiza la entrada 3 de la tabla de página de Pi con los valores: 1 para el bit

P, 0 para el bit M, se asigna 1 al campo MP y se retira de LML el marco 1. Después de estas acciones se deberá relocalizarse la dirección 3098 en su dirección real tomando en cuenta que esa página se cargó en el marco de página 1.

En el ejemplo de la figura 74 se ha supuesto que hay un marco de página libre pero si eso no es cierto deberá quitarse de memoria alguna página (denominada víctima) para poner en su lugar la página solicitada.

```
/*Algoritmo Demanda de página. Recibe una dirección lógica.
Utiliza las estructuras de datos: tabla de página (TP) y lista de marcos libres (LML) de la figura 2.19*/
DemandaPagina(Direc)
   Calcular el par (Pag, D)            //Donde Pag es la pagina y D el desplazamiento dentro de la página
   Si TP->Pag.P == 1                   //Si esta condición es verdadera …
      Entonces                          // … la página Pag está en memoria
         Acciones adicionales           //Las acciones dependen del algoritmo de reemplazamiento
      En caso contrario                 //Hay una falta de página
         Si LML no es vacía             //Si existe algún marco de página libre
            Entonces
               Tomar un marco M         //Se toma cualquier marco (todos son iguales)
               Cargar Pag en marco M
               Eliminar M de LML        //el marco M ya no está libre
               Acciones adicionales     //Las acciones dependen del algoritmo de reemplazamiento
            En caso contrario           //Se necesita un reemplazamiento de página
               Seleccionar V            //Selecciona página V como víctima (depende del algoritmo)
               Si TP->V.M == 1          //Si esta condición es verdadera …
                  Entonces              // … la página V ha sido modificada
                     Salvar V           //Copiar V hacia el equipo de resguardo porque ha sido modificada
               Cargar P en marco V
               Acciones adicionales     //Las acciones dependen del algoritmo de reemplazamiento
   //Acciones generales
   Actualizar la tabla de página.
   Calcular dirección real usando el par (Pag, D) y la tabla de página (TP)
   Retornar la dirección calculada.
Fin
```

Figura 74. Algoritmo general para la demanda de página.

La figura 74 presenta un algoritmo general para satisfacer demanda de páginas que no hace referencia a ningún algoritmo de reemplazamiento específico, por eso será necesario realizar algunos ajustes cuando se utilice alguno en particular, lo que puede implicar (en algunos casos) ajustes a la estructura de la tabla de página.

2.6.2. Memoria virtual segmentada

La segmentación permite que los programadores vean la memoria organizada en distintos espacios de direcciones contiguas y de diferentes tamaños conocidos como segmentos. Las referencias a memoria están formadas por pares que toman la forma: (segmento, desplazamiento).

Entre las ventajas de esta forma organizativa, desde el punto de vista del programador, se pueden citar las siguientes:

- Se simplifica el trabajo con estructuras de datos que crecen dinámicamente, para lo cual el SO puede mover los segmentos que las contienen a un área de mayor longitud o desalojarlos para volverlos a cargar.
- Se pueden modificar y recompilar las partes (programas) de un sistema (conjuntos de programas) sin necesidad de enlazar y recargar el sistema completo.
- Resulta más fácil compartir las partes de un proceso (código o datos).
- Resulta más fácil establecer mecanismos de protección que se asocian a los segmentos.

La segmentación tal y como fue analizada en el epígrafe 2.5 se apoya en una tabla de segmentos asociada a cada uno de los procesos, cada entrada en esa tabla tiene dos campos: dirección de inicio del segmento y longitud del segmento.

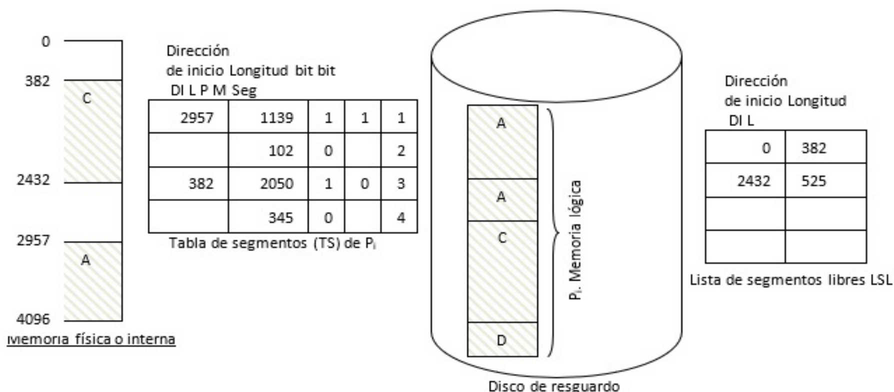


Figura 75. Segmentado con memoria virtual.

La figura 75 muestra la idea de la memoria virtual segmentada. En la tabla de segmentos se pueden observar (de izquierda a derecha): los campos Dirección de inicio (DI) y longitud (L), que contienen la dirección donde comienza el segmento y su tamaño respectivamente; el bit P o de presencia, para señalar cuando un segmento está en la memoria física (segmentos 1 y 4 en el ejemplo); el bit M o de modificación para identificar los segmentos que han sido modificados (segmento 1 en el ejemplo) y el campo Seg para identificar los segmentos del proceso (1,

2, 3 y 4, en este caso). En este esquema cuando se necesita acceder a una dirección de memoria deberá hacerse la traducción de la dirección virtual (segmento, desplazamiento) a la dirección real o física, para lo cual se usa la tabla de segmentos.

Las tablas de segmentos son de longitud variable y por eso no es posible cargarlas en los registros del procesador, pero sí es posible mantenerlas en memoria principal para agilizar los accesos y hacer que un registro apunte a su dirección de inicio.

```
/*Algoritmo Demanda de segmentos. Recibe número del segmento, Seg, y el desplazamiento, D.
Utiliza la estructura de datos tabla de segmentos página (TS) y lista de segmentos libres (LSL) de la figura 2.21*/
DemandaSegmento(Seg, D)
   Si TS->Seg.P == 1                      //Si esta condición es verdadera …
      Entonces                            // … el segmento Seg está en memoria
         Acciones adicionales             //Las acciones dependen del algoritmo de reemplazamiento
      En caso contrario                   //Hay una falta de segmento
         Buscar un LSL.L >= TS->Seg.L     //Si existe algún espacio libre que pueda alojar el segmento
            Entonces
               Cargar Seg a partir de LSL.DI
               Actualizar LSL             //Ajusta la entrada de LSL al espacio que, posiblemente, haya quedado
               Acciones adicionales       //Las acciones dependen del algoritmo de reemplazamiento
            En caso contrario             //Hay un reemplazamiento de segmento
               Seleccionar (V de TS tal que:
                TS->Seg.L >= Seg)         //La víctima tiene que ser mayor o igual al segmento que se cargará
               Si TS->V.M == 1            //Si esta condición es verdadera …
                  Entonces                // … el segmento V ha sido modificado
                     Salvar V             //Copiar V hacia el equipo de resguardo porque ha sido modificado
               Cargar Seg a partir de TS->V.DI
               Actualizar LSL             //Ajusta la entrada de LSL al espacio que, posiblemente, haya quedado
               Acciones adicionales       Las acciones dependen del algoritmo de reemplazamiento
   //Acciones generales
   Actualizar la tabla de segmentos.
   Calcular dirección real usando el par (Seg, D) y la tabla de segmento (TS)
   Retornar la dirección calculada.
Fin
```

Figura 76. Algoritmo general para la demanda de segmentos.

El algoritmo de la figura 76 muestra la idea general para atender demanda de segmentos y es una adaptación de la figura 76 que toma en cuenta los aspectos siguientes:

- Los segmentos solo pueden cargarse en espacios libres mayores o iguales que su longitud.
- Solo son candidatos a segmentos víctimas los que tienen una longitud mayor o igual que el segmento que se cargará en memoria (Lezcano Brito, 2018).
- Cuando se cargue un segmento en un espacio mayor que su longitud, será necesario situar, en la lista de espacios libres, una referencia al espacio libre que quede.

- Pueden necesitarse algunas operaciones de ajustes a la lista de espacios libres cada vez que se actualice, lo que depende de su implementación.

2.6.3. Uso de información común. La protección

Para que varios procesos puedan usar una misma página basta con que se apunte a ella desde dos (o más) tablas de páginas. El código de las partes compartidas tiene que ser reentrante, o sea no puede modificarse mientras se esté ejecutando, debido a que la modificación hecha por un proceso Pi afectará directamente a otro proceso Pj que está usando el mismo código (siendo i, j cualesquiera).

En la figura 77 se muestra un sistema paginado que esquematiza la idea. Debe observarse que las tablas de páginas de los procesos P1 y P2 hacen referencia a los marcos de páginas 0 y 2 donde reside el código de un sistema S que usan ambos procesos (observe las tramas diferentes para resaltar la idea).

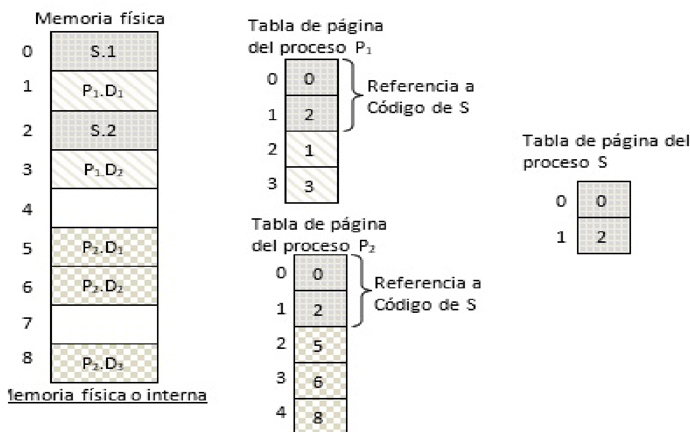


Figura 77. Páginas compartidas.

En ese caso no es necesario que cada proceso cargue el sistema S debido a que ellos solo tendrán distintos datos, pero comparten el mismo código, de manera que una vez que se cargue S en memoria las ordenes posteriores para usarlo deberán tener las mismas entradas para referirse a S y entradas adicionales para sus datos. Ejemplos de sistemas que se pueden usar en esta forma son: los editores, los compiladores y los sistemas de bases datos entre otros.

La misma idea se usa en la segmentación para compartir segmentos, en este caso los segmentos compartidos se referencian desde la tabla de segmentos. Cada entrada incluye la longitud y la dirección de inicio de cada segmento resulta más fácil implementar políticas para proteger la información (tanto código como datos); en este caso los procesos no pueden violar las fronteras de acceso debido a que cada dirección generada se compara con el límite establecido por el campo longitud de la tabla de segmentos.

En el paginado se puede usar el mecanismo presentado en el párrafo anterior, pero en ese caso la estructura de páginas del programa no es visible para el programador lo que dificulta las especificaciones para compartir y proteger la información.

Sistemas combinados

Como se ha podido apreciar tanto el paginado como el segmentado tienen sus puntos débiles y sus fortalezas, por ese motivo algunos sistemas ofrecen combinaciones de ambas técnicas para tratar de aprovechar las ventajas de cada uno. En esos casos el procesador tiene que tener la capacidad para hacerlo y el SO debe diseñarse para explotar esa facilidad.

En los sistemas paginados/segmentados el programador divide el espacio de direcciones de usuarios en segmentos. Cada uno de los segmentos se divide en páginas (de igual longitud) que tienen el mismo tamaño de los marcos de páginas de la memoria. Los segmentos con longitudes inferiores al tamaño de la página deberán ocupar una página completa.

Desde el punto de vista del programador una dirección lógica se mantiene en la forma del par (segmento, desplazamiento). Desde el punto de vista del SO, el desplazamiento del segmento se ve como un número de página y un desplazamiento de página, para una página dada, dentro del segmento especificado.

Políticas de manejo de la memoria virtual

Las políticas de manejo de memoria virtual establecen cuando una pieza de código o de datos debe traerse a memoria, En ese sentido existen dos estrategias nombradas paginado puro y prepaginado. Estos

son los usados en la literatura, pero en realidad hacen referencia a cualquier pieza de memoria virtual (segmento, página o cualquier otra).

Paginado puro

En este caso las piezas (página, segmento, entre otros.) solo se traen a memoria cuando se demandan, es decir cuando se hace referencia a alguna de sus localizaciones, lo que provoca una considerable cantidad de faltas de acceso a memoria cuando el proceso se ejecuta por primera vez.

El hecho de que las páginas se carguen por demanda apoya el principio de localidad, por ese motivo se espera que después de haberse cargado un conjunto de piezas dado exista una mayor probabilidad de que las referencias a memoria no provoquen faltas de acceso.

Prepaginado

La mayoría de los equipos de memoria secundaria tienen dos momentos de acceso: el tiempo de posicionamiento (seek time) y el tiempo de latencia (latency time).

El primero se refiere al tiempo que tarda el cabezal de lectura/escritura en situarse en la posición longitudinal donde están los datos (el cilindro en un disco duro), mientras el segundo es el tiempo que tiene que esperarse para que el bloque de datos que se desea transferir esté debajo del cabezal. Debido a estas características resulta más eficiente traer a memoria un conjunto de piezas contiguas del disco que cargarlas separadamente.

Cuando se inicia un proceso, usando técnicas de prepaginado, se carga un subconjunto de sus piezas (páginas, segmentos, entre otros.) que están almacenadas en forma contigua. En este caso se tiene la esperanza de que las futuras referencias a direcciones también estén dentro del subconjunto cargado, aunque en realidad no tiene que cumplirse este deseo.

Políticas de reemplazamiento

En las figuras 76 y 77 se presentaron seudocódigos para satisfacer la demanda de página y de segmentos respectivamente, se dejó una brecha abierta en ellos para poder ajustarlo a algoritmos de reemplaza-

miento específicos, algunos de los cuales se presentan en esta sección. Se sugiere modificar los seudocódigos de ambas figuras para ajustarlos a estos algoritmos.

Reemplazamiento óptimo

Este algoritmo mira hacia el futuro y selecciona como víctima a la unidad de memoria (hace referencia a una página o un segmento) que más va a tardar en referenciarse, queda claro que esa estrategia provoca la menor cantidad de faltas de acceso posible y ese sentido se nombra óptimo. Infortunadamente no es posible predecir el futuro y por eso no puede implementarse tal y como es, pero se utiliza como referencia para realizar comparaciones y también se implementan algunas aproximaciones a él.

El primero en llegar será el primero en salir (FIFO). Este es un algoritmo muy sencillo debido a que simplemente escoge como víctima a la unidad (página, segmento, entre otros.) que lleva más tiempo en memoria.

La idea de esta estrategia asume que, si una unidad de memoria se ha usado por bastante tiempo, se tardará mucho en usarse en el futuro o ya no se usará más. El funcionamiento es el siguiente: existen ocho colas donde llegan los paquetes para ser procesados, y estos se enumeran según su orden de llegada (Ap & Annoni Pazeto, 2017).

Como lo que interesa conocer es cuál es la pieza más vieja y no su edad, las páginas pueden insertarse al final de una cola cuando entran a memoria, así la más vieja siempre estará en la cabeza de la cola (Lezcano Brito, 2018).

El algoritmo FIFO es fácil de comprender y programar, pero su rendimiento no es bueno y sufre la anomalía de Beladay (Lezcano Brito, 2018). Esta anomalía hace que, a veces, existan más faltas de páginas cuando mayor es la cantidad de marcos que se pueden referenciar desde la tabla de página.

La unidad menos recientemente usada (LRU). Este algoritmo, no trata de predecir el futuro (como el óptimo), en su lugar mira al pasado para sustituir la unidad de memoria que lleva más tiempo si referenciarse. En este caso se asume una actitud optimista al pensar que si hace tiempo no se utiliza una unidad dada, la probabilidad de usarla en un futuro

cercano es menor y por ese motivo se toma como víctima.

Se utiliza un bloque que no ha sido referenciado ya en algún tiempo, a este algoritmo se le asocian contadores a las líneas de caché para saber cuándo es un acierto o un fallo, cabe destacar que un acierto sucede cuando se hace referencia a un marco de bloque (Vázquez Carmona, et al., 2019).

Para implementar LRU resulta necesario asociar a cada unidad de memoria el momento en que se usa, de manera que se pueda elegir como víctima la que lleva más tiempo inactiva. Esta acción hace que el algoritmo sea más costoso que el óptimo y el FIFO.

La política LRU se usa con bastante frecuencia y, en términos generales, se considera una buena estrategia. La principal dificultad se encuentra en su implementación debido a que necesita un apoyo sustancial del *hardware* para determinar el momento en que se usa cada unidad de memoria, es posible hacer dos implementaciones:

- Usando contadores. En este caso a cada entrada en la tabla de página o de segmento se asocia un campo para controlar el tiempo de uso y se usa un registro de la CPU para mantener un reloj lógico que se incrementa cada vez que ocurra una referencia a cualquier página.

Cada vez que se haga una referencia a una página dada el valor del reloj lógico se copia en el campo tiempo de uso de la entrada en la tabla de página (o de segmento) correspondiente. De esta manera la unidad de memoria a sustituir será aquella que tenga ese campo con el menor valor.

- Con una lista doblemente enlazada con apuntadores al inicio y al final. En este caso cada vez que se use una unidad de memoria, su número se inserta al inicio de la lista, de manera que en la cabeza de la lista estará la más recientemente usada y en la cola la menos recientemente usada.

Aproximación a LRU

Muchas computadoras no proveen suficiente apoyo de *hardware* para implementar LRU, pero brindan un bit adicional en la tabla de página (o de segmento) que puede usarse como bit de referencia para implementar LRU a través de micro código o *software* en la forma siguiente:

- Cuando se crea un proceso nuevo, el SO pone todos los bits de referencia en 0 (ninguna página ha sido usada).
- Cuando un proceso usa una unidad dada, al bit de referencia de esa unidad en la tabla de página (o de segmento) se le asigna 1 (por *hardware*) con lo cual se conoce cuáles unidades se han usado, pero no el orden de uso.

Los SO y la memoria virtual

Como ya se ha establecido, los SO manejan los recursos de *hardware* pero a la vez no pueden hacer grandes cosas si este componente no le proporciona las herramientas adecuadas, eso es precisamente lo que ocurrió con los primeros SO UNIX que no ofrecían el manejo de memoria virtual debido a que los procesadores para los que se implementaron no soportaban paginado ni segmentado; por otra parte, la mayoría de los SO modernos (incluido UNIX por supuesto) soportan la memoria virtual, generalmente en su forma paginada.

El manejador de memoria virtual es el responsable de asignar las páginas y controlar el paginado, puede operar sobre varias plataformas con páginas que fluctúan entre 4 y 64 Kb.

Los SO Windows implementan la memoria virtual paginada con agrupamiento (clúster), en esta técnica cuando ocurre una falta de página se trae a memoria un conjunto de páginas que son vecinas a ellas.

Cuando se crea un proceso se le asignan dos conjuntos:

- Conjunto de trabajo mínimo (se denominará CTMin). Formado por la cantidad mínima de páginas cargadas en memoria que se le garantiza a un proceso.
- Conjunto de trabajo máximo (se nombrará CTMax). Establece la cantidad máxima de páginas que se le puede asignar a un proceso cuando hay suficiente memoria.

El manejador de memoria virtual mantiene una lista de marcos de páginas libres (se le llamará LML) a la cual se asocia un valor que indica si hay suficiente memoria libre (se nombrará SM).

- Las faltas de página ocasionadas por procesos que están por debajo de su CTMax se resuelven cargando la página solicitada en los marcos de páginas referenciados en LML.

- Las faltas de página ocasionadas por procesos que ya están en su CTMax se resuelven sustituyendo alguna página del proceso (página víctima) que se escoge usando la política LRU. El reemplazamiento es local, o sea la víctima se selecciona entre las páginas del proceso.

Cuando la cantidad de memoria libre está por debajo de SM, el manejador de memoria explora los CTMin de los procesos, con el fin de retirarles páginas a aquellos que tienen asignada una cantidad de páginas mayor a CTMin, este proceder se repite hasta que se restablezca el valor de SM; la tarea se hace en los dos espacios de memoria, es decir en el espacio de usuario y el del SO.

Los SO de la familia Windows usan un archivo especial como equipo de intercambio (swapping) para el paginado. Puede definirse un archivo de intercambio para cada disco (físico o lógico) o puede tenerse un solo archivo global que lo utilizará todo el sistema de cómputo. Para ver esos detalles en el ambiente de trabajo de Windows, puede accederse a:

Panel de control|Sistema|Configuración avanzada del sistema

La acción anterior mostrará una ventana como la de la figura 70, si se oprime el botón configuración de la sesión Rendimiento, se mostrará otra ventana, figura 78, en la que se ha escogido la pestaña Opciones avanzadas.

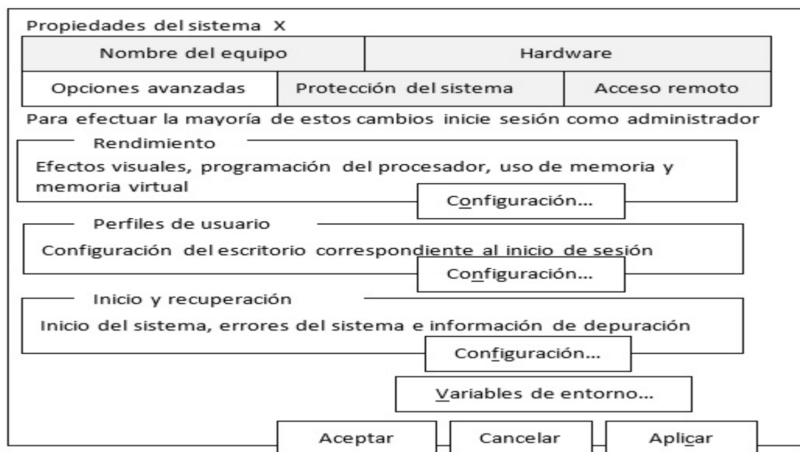


Figura 78. Windows 10. Configuración avanzada del sistema.

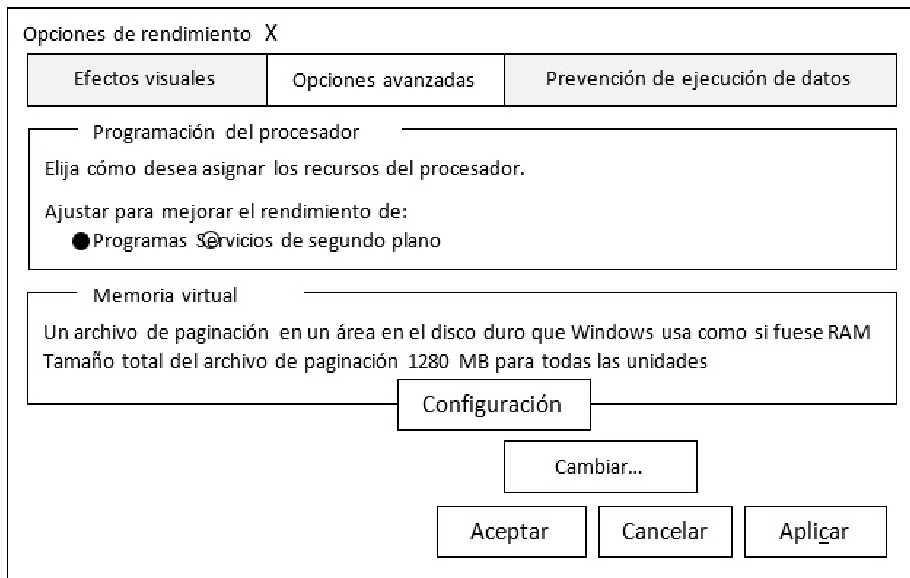


Figura 79. Windows 10. Opciones de rendimiento.

Puede observarse que el programador puede elegir entre un rendimiento del procesador ajustado a los programas (capítulo I) o a los servicios de segundo plano.

La sesión Memoria virtual permite establecer el tamaño que tendrá el archivo de paginación para lo cual se usa el botón Cambiar, debe tenerse alguna experiencia para establecer valores adecuados a cada sistema de cómputo particular y a las aplicaciones que se estén ejecutando.

Manejo de la memoria virtual en el SO UNIX

Las primeras versiones de UNIX no utilizaban técnicas de memoria virtual y se basaban en la estrategia MVT. Las implementaciones actuales usan memoria virtual paginada y se trazan la meta de ser independientes de las máquinas sobre las que están soportados. Con ese propósito definen las estructuras de datos que se analizan a continuación.

Tabla de página

Cada entrada en una tabla de página UNIX tiene los campos siguientes

Número del marco	Edad	Copiar en escritura	Modificada	Referenciada	Válida	Protegida

Figura 80. UNIX. Entrada en la tabla de página.

- Número del marco. Contiene el marco de página (memoria real) en que está cargada la página (memoria lógica).
- Edad. Marca el tiempo que ha permanecido la página en memoria sin ser referenciada. Este campo es dependiente del procesador
- Copiar en escritura. Especifica si la página se comparte entre varios procesos. Cuando un proceso escribe sobre ella será necesario copiar la página en todos los procesos que la comparten.
- Modificada. Indica si la página se modificó.
- Referenciada. Indica que la página ha sido referenciada. Tendrá el valor 0 cada vez que se cargue la página en memoria.
- Válida. Indica si la página está en memoria.
- Protegida. Indica si se permiten operaciones de escritura sobre la página.

Descriptor de bloque de disco

Especifica el lugar del disco donde se localiza cada página y posee los campos siguientes (figura 81):

Número del equipo de intercambio	Número del bloque	Tipo de almacenamiento

Figura 81. UNIX. Descriptor de bloque de disco.

- Número del equipo de intercambio (swap). Contiene la identificación numérica del equipo en que está alojada la página, lo que permite utilizar más de un e1quipo con ese propósito.
- Número del bloque. Localización del bloque de disco donde reside la página.
- Tipo de almacenamiento. Puede ser de dos tipos: una unidad de intercambio (swap), un archivo ejecutable; en este último caso habrá una indicación de si es necesario que el SO limpie (ponga en 0) las localizaciones del marco de página antes de cargar el primer bloque de programa o dato.

Tabla de marcos de página

Cada entrada en una tabla de marcos de página tiene los campos siguientes (figura 82):

Estado de la página	Contador de referencias	Equipo lógico	Número de bloque	Puntero

Figura 82. UNIX. Entrada en la tabla de marcos de página.

- Estado de la página. Indica si el marco de página está libre o está ocupado por alguna página. En este último caso se especifica su estatus: sobre el equipo de resguardo, en archivo ejecutable, en progreso de DMA.
- Contador de referencias. Cantidad de procesos que referencian la página.
- Equipo lógico. Equipo lógico que contiene una copia de la página.
- Número de bloque. Localización de la copia de la página sobre el equipo lógico.
- Puntero. Puntero a otra tabla de marcos de páginas.

Tabla de intercambio (swap)

Cada entrada en una tabla de intercambio contiene los campos siguientes (figura 83):

Contador de referencias	Número de l página

Figura 83. UNIX. Entrada en la tabla de intercambio.

- Contador de referencias. Cantidad de entradas de la tabla de página que apuntan a la página que está sobre el equipo de intercambio (swap).
- Número de la página. Identificador de la página sobre la unidad de almacenamiento.

El manejo de la memoria virtual en otros SO se implementa con técnicas diferentes, aunque todas tratan de minimizar el tráfico de páginas. No es objetivo de este texto hacer una exploración sobre cada una de estas particularidades y solo se ha mencionado, someramente, la forma en que lo hace Windows y los SO tipo Unix.

Archivos especiales relacionados con la memoria en los SO tipo UNIX

A diferencia de Windows, UNIX utiliza una partición de disco o un disco como equipo de resguardo o intercambio (swap) para manipular la memoria virtual y mantiene un conjunto de archivos que brindan información importante acerca de diversos detalles del sistema.

En el directorio /proc se localizan varios de los archivos que se mencionaron en el párrafo anterior, en particular los archivos meminfo (figura 84), pagetypeinfo, vmstat y zoneinfo contienen información acerca de la memoria.

```
$cat /proc/meminfo | less
MemTotal: 1026488 kB
MemFree: 604348 kB
Buffers: 66260 kB
Cached: 181800 kB
SwapCached: 0 kB
Active: 195068 kB
Inactive: 181656 kB
Active(anon): 129032 kB
Inactive(anon): 5200 kB
Active(file): 66036 kB
Inactive(file): 176456 kB
:
```

Figura 84. UNIX. Visualización del archivo meminfo.

Puede usarse el comando cat o cualquier editor de texto, para analizar los contenidos de los archivos mencionados anteriormente. El lector deberá hacer esa labor para tratar de comprender todos los detalles (son muy diversos).

La figura 84 muestra el uso combinado de los comandos cat y less, que se enlazan a través de una tubería (|) para que el primero envíe sus resultados al segundo, el cual los filtrará y los mostrará pantalla a pantalla.

Los SO manejan diversos recursos de *hardware*, la memoria es uno de los más importantes debido a que todo proceso, incluido el propio SO, necesita estar cargado en ella (parcial o totalmente) para ejecutarse, de hecho, para dejar de ser un programa y convertirse en proceso.

La memoria se puede ver como un arreglo de bytes o palabras de localizaciones, cada una de ellas tiene una dirección que se utiliza para acceder a su contenido.

Los primeros SO eran monoprogramados y por eso el esquema de memoria era muy sencillo. Este recurso se dividía en dos partes: una para el SO y otra para el único programa que se podía ejecutar.

El surgimiento de la multiprogramación hizo más complejo el esquema para manipular la memoria, debido a que en este caso se tienen varios procesos en memoria y por eso hay que buscar una manera efectiva de repartirla a la vez que se protegen las zonas de cada proceso.

Los primeros SO, incluidos los multiprogramados, cargaban en memoria todo el código y los datos de los procesos a ejecutar. Con ese fin se diseñaron los sistemas MFT y MVT, pero con el tiempo se observó que bastaba con cargarlos parcialmente, lo que dio pie a las técnicas de memoria virtual.

Para implementar las técnicas de memoria virtual tiene que existir una copia de los procesos en memoria externa (casi siempre un disco), desde allí entran y salen las partes que componen los procesos; el componente de *hardware* que se usa con ese propósito se le ha llamado equipo de resguardo en este texto.

La técnica de memoria virtual más usada hoy en día es el paginado, en la cual la memoria física del equipo de cómputo y la memoria lógica de cada proceso se dividen en unidades de igual longitud, denominadas marcos de página y páginas respectivamente. Para implementar el paginado se usan tablas de páginas (una por cada proceso), estas tablas permiten mapear las direcciones lógicas de los procesos en sus direcciones físicas y también conocer otros detalles para evitar el tráfico de página. Los sistemas paginados sufren fragmentación interna.

Otra técnica de memoria virtual muy difundida es el segmentado, en este caso no existe una división a priori de la memoria física, pero la memoria lógica de cada proceso se divide en unidades, denominadas segmentos. Los segmentos se forman de acuerdo al objetivo del código o de los datos; por ejemplo, una función para hacer algo, una matriz de adyacencia, entre otros. Para mapear las direcciones lógicas en direcciones físicas se usan tablas de segmentos. Los esquemas segmentados sufren fragmentación externa.

Cuando se usa memoria virtual y no hay espacio para almacenar una parte que se referencia, es necesario situarla en el lugar de otra que está en memoria o sea reemplazarla. Existen diversos algoritmos con ese propósito, entre ellos se pueden mencionar LRU y FIFO.

2.7. Ejercicios propuestos

1. **Las particiones fijas y las variables son técnicas de administración de la memoria. Haga dos tablas que reflejen:**

a) Las semejanzas entre ellas.

b) Las diferencias entre ellas.

2. **Explique qué es la memoria virtual y cómo esta técnica hace ver la memoria mayor de lo que es realmente.**

3. **Explique los siguientes conceptos dejando bien claro las diferencias entre ellos:**

a) Memoria principal o física.

b) Memoria secundaria.

c) Memoria lógica.

4. **El paginado es la técnica más usada para manejar la memoria virtual.**

a) Explique las estructuras de datos que se usan para implementar esta técnica.

b) Explique qué es la fragmentación interna y por qué el paginado sufre de ella. Apoye su explicación con un ejemplo.

c) Explique cómo se pueden compartir páginas cuando se usa esta técnica.

5. **El segmentado es una técnica de manejo de memoria virtual que ofrece una visión de la memoria más cercana al programador ¿En que se basa esta afirmación?**

a) Explique las estructuras de datos que se usan para implementar esta técnica.

b) ¿Por qué se dice que es más difícil de implementar que el paginado?

c) Explique qué es la fragmentación externa y por qué el segmentado sufre de ella. Apoye su explicación con un ejemplo.

d) Explique cómo se pueden compartir segmentos cuando se usa esta técnica ¿Por qué se dice que es más fácil ofrecer mecanismos de protección en el segmentado que en el paginado?

6. **Explique las políticas de manejo de memoria virtual conocidas por paginado puro y prepaginado. Resalte ventajas y desventajas y justifique su uso.**

a) El algoritmo de reemplazamiento FIFO sufre la anomalía de Beladay. Suponga dos configuraciones para paginado puro:

- **i. En la primera la tabla de página solo permite alojar 4 referencias a marcos de páginas. Muestre una cadena de demandas de páginas donde se aprecie el funcionamiento del algoritmo.**
- **ii. En la segunda la tabla de página solo permite alojar 6 referencias a marcos de páginas. Use la misma cadena de demandas de páginas del inciso anterior y verifique si se produce la anomalía de Beladay.**

7. **Explique los algoritmos de reemplazamiento estudiados, resaltando sus ventajas y desventajas, así como los cambios que deben hacerse (si es que es necesario) a la tabla de página o de segmentos para poder implementarlos:**

a) Reemplazamiento óptimo.

b) FIFO.

c) LRU.

8. **Además de los algoritmos de reemplazamiento anteriormente mencionados, existen otros que no se analizaron en este texto. Búsquelos en bibliografía complementaria y analícelos.**

9. **En la figura 27 se presentó un algoritmo general para atender la demanda de página. Modifique ese algoritmo, y las estructuras de datos necesarias, para ajustarlo a los algoritmos de reemplazamiento FIFO y LRU respectivamente.**

10. **En la figura 29 se presentó un algoritmo general para atender la demanda de segmentos. Modifique ese algoritmo, y las estructuras de datos necesarias, para ajustarlo a los algoritmos de reemplazamiento FIFO y LRU respectivamente.**

11. **El tráfico de páginas (o de segmentos) es un problema indeseado.**

a) Explique qué es.

b) El principio de localidad puede usarse para disminuir el efecto negativo del tráfico de páginas. Explique este principio y diga por qué se hace la afirmación anterior.

12. 12. Para poder usar la memoria virtual es necesario que la información manejada (código o datos) sea relocalizable.

a) Explique qué significa que sea relocalizable.

b) ¿Por qué no se pueden usar las técnicas de memoria virtual sin la relocalización?

13. Cuando se utiliza información (código o datos) común o sea cuando varios procesos utilizan un mismo sistema; por ejemplo, un editor, es necesario que la parte compartida sea reentrante.

a) Explique qué significa que sea reentrante.

b) Justifique esta restricción.

Capítulo III. Equipos de almacenamiento masivo

3.1. El sistema de archivos

La mayoría de los datos que manipulan los procesos, así como sus salidas, residen sobre algún soporte de almacenamiento externo que es manejado por el sistema de archivos.

En general el sistema de archivo puede verse como la integración lógica de tres partes:

- Al más alto nivel se encuentra una interfaz con la cual interactúan los usuarios y los programadores y es la parte visible para los usuarios.
- En un nivel intermedio existen diversas estructuras de datos y algoritmos manejados por el SO.
- Al nivel más bajo se encuentran los equipos externos que sirven de soportes de almacenamiento masivo.

Los medios que se conectan a cualquier equipo de cómputo son muy diversos y pueden tener múltiples propósitos; por ejemplo, los usuarios se comunican con los equipos de cómputo a través de terminales (monitor-teclado), ratones, entre otros., utilizan impresoras para imprimir información que está guardada en diversos equipos (discos, cintas, entre otros.), se comunican con otros lugares a través de módems, tarjetas de redes, entre otros.

Toda esa amalgama de equipos posee diferentes estructuras internas y diversas formas organizativas, pero el SO debe brindar una interfaz cómoda para usar el equipamiento sin que el usuario tenga que preocuparse por detalles que solo les interesan a los especialistas.

Este capítulo se dedica a los equipos de almacenamiento masivo y constituye una lectura obligada para poder comprender el próximo, que abundará acerca del sistema de archivo.

3.2. Los discos magnéticos

Los discos magnéticos son un soporte invaluable para almacenar información; sobre los discos duros, en particular, recae el peso mayor del almacenamiento secundario proporcionado por los equipos de cómputo actuales.

La figura 85 presenta una imagen esquematizada de la estructura in-

terna de un disco duro, los cuales están formados por un conjunto de platos que tienen dos superficies regrabables, cada una de ellas dividida lógicamente en pistas concéntricas circulares, las cuales a la vez se dividen en unidades denominadas sectores.

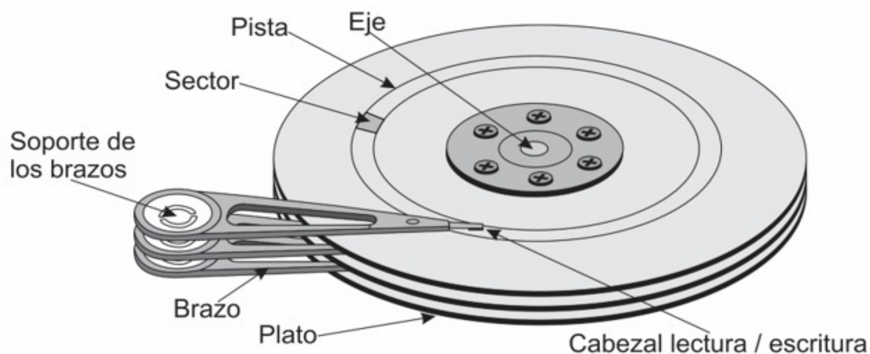


Figura 85. UNIX. Vista esquemática de un disco duro.

Todos los platos giran a la vez, en contra de las manecillas del reloj, movidos por un motor asociado al eje central que los une. El disco incluye un soporte que contiene un brazo, con un cabezal de lectura/escritura por cada superficie. El conjunto de todas las pistas accesible por un posicionamiento del brazo recibe el nombre de cilindro por su similitud con esa figura geométrica.

Cuando el disco se está usando el motor rota a alta velocidad; por ejemplo, 5,400, 7,200, 10,000 y 15,000 rpm. La velocidad de acceso a los contenidos del disco está determinada por dos factores:

- El tiempo de posicionamiento o tiempo de acceso aleatorio que está formado por dos partes:
  - El tiempo que se consume para mover el brazo del disco desde la posición que ocupa hasta el cilindro deseado. Se denomina tiempo de búsqueda (seek time).
  - El tiempo que debe esperar el cabezal de lectura/escritura para que la rotación del disco ponga el sector deseado debajo de él. Se denomina tiempo de latencia o de rotación.
- El rango de transferencia que viene dado por el tiempo que se demoran los datos en fluir entre el disco y la computadora.

En un SO multiprogramado el SO mantiene una cola por cada equipo de E/S, de manera que un disco puede tener múltiples peticiones de lectura y escritura que se originan desde diversos procesos. Esas colas deben planificarse en aras de que las peticiones se satisfagan de forma adecuada.

Se conoce como el ancho de banda del disco al total de bytes transferidos dividido entre el total de tiempo transcurrido entre la primera petición y la terminación de la última transferencia, la idea entonces es tratar de “optimizar” esos valores.

Cada vez que un proceso necesita una E/S de disco deberá hacer una llamada al SO especificando la información siguiente:

1. El tipo de operación (entrada o salida).
2. La dirección del dato en el disco.
3. La dirección del dato en la memoria.
4. La cantidad de sectores a transferir.

Obsérvese que el hecho de hacer una petición no significa que será atendida de inmediato debido a que el disco puede estar ocupado con peticiones anteriores y en ese caso la nueva petición deberá quedar en una cola que puede tener otras peticiones pendientes. A partir de esta realidad queda clara la necesidad de usar algún algoritmo para decidir cuál de las peticiones pendientes será la próxima a satisfacer.

3.2.1. Algoritmos de planificación de discos magnéticos

Existen varios algoritmos para planificar discos, ellos pueden tomar en cuenta la posición en que está el sector deseado, o el momento en que se hace una petición dada.

El primer algoritmo que se analiza es el más sencillo de todos y se denomina “el primero en llegar es el primero en servirse”, más conocido por sus siglas en inglés FCFS. El algoritmo se basa en el orden de llegada de las peticiones y no proporciona las mejores respuestas con relación a la velocidad.

Para analizar el algoritmo de una manera más práctica se hace uso

del ejemplo presentado en la figura 86. Considérese que en la cola de peticiones de E/S para un disco dado se tiene la siguiente secuencia de peticiones (los números representan cilindros): 10, 190, 40, 160, 300, 130, 210, 5 y que el brazo de los cabezales está en ese momento en el cilindro 70.

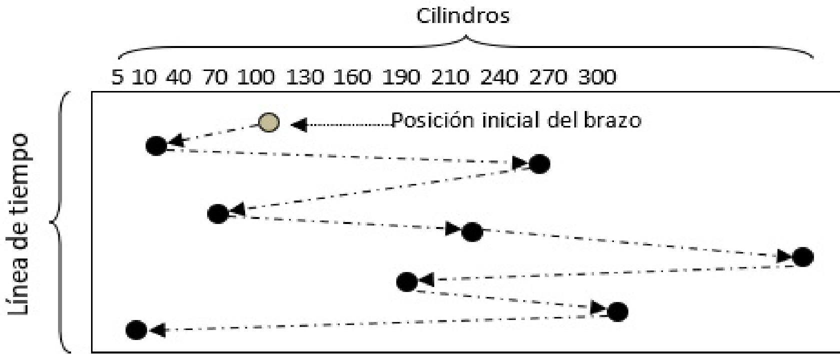


Figura 86. Movimientos del cabezal con FCFS.

En la figura se refleja el movimiento en zigzag y la cantidad de movimientos del brazo que provoca esta secuencia de peticiones. Se puede inferir que el comportamiento del algoritmo no es predecible debido a que dependerá del orden de las llegadas de las solicitudes, obsérvese que el brazo puede moverse de un extremo al otro del disco para satisfacer la petición actual, a pesar de que puede existir una petición que esté cercana a él pero que arribó más tarde. El siguiente algoritmo se basa en esa última observación.

El segundo algoritmo intenta hacer la menor cantidad posible de movimientos del brazo. Para lograr su objetivo toma en cuenta la distancia que hay desde la posición del cabezal hacia cada una de las peticiones; por ejemplo, en la cadena analizada anteriormente se podría satisfacer la petición del cilindro 40 después de la del 10 y no ir hasta el cilindro 190, como hizo FCFS. Debido a ese razonamiento al algoritmo se le denomina "el tiempo de posicionamiento más corto primero", frecuentemente referenciado por sus siglas en inglés SSTF (Shortest Seek Time First).

Aunque SSTF se porta mejor que FCFS, no es el más eficiente y puede provocar inanición a las peticiones que estén más lejanas. Observe la

figura 87, en la que existe una concentración de peticiones cercanas al centro del disco, si el brazo arriba a esa zona la probabilidad de satisfacer las peticiones de los cilindros exteriores es baja y se reduce si siguen arribando peticiones hacia el centro.

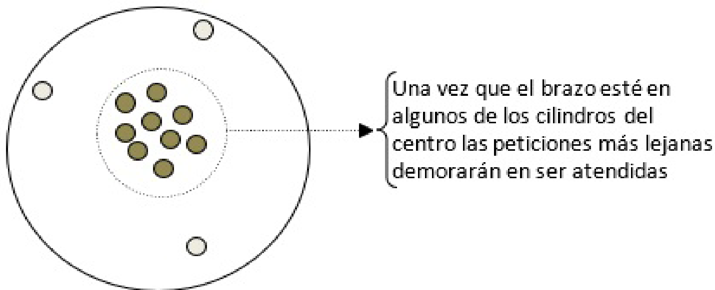


Figura 87. SSTF privilegia las peticiones cercanas.

Una manera "más natural" sería mover el brazo en una dirección dada (hacia adelante o hacia atrás) para ir satisfaciendo las peticiones que se encuentren en el camino, tal y como hacen los elevadores en los edificios. Debido a esa forma de dar servicio el algoritmo que se analiza a continuación recibe el nombre del Elevador.

En este caso el brazo comienza situándose al final del disco y se mueve hacia el extremo opuesto satisfaciendo todas las peticiones que encuentre, una vez que alcance el extremo opuesto regresa hacia el otro extremo satisfaciendo las peticiones que han arribado mientras se movía en dirección contraria. El algoritmo se conoce como Scan en la literatura anglófona.

Una variante de algoritmo del Elevador o Scan que pretende tener mejores tiempos de respuesta es el Elevador circular (C-Scan). En este caso el "elevador" se mueve hacia una dirección y cuando llega al final no regresa dando servicios, sino que retorna directamente al lugar de inicio (el final o el inicio del disco) para de esa forma no retardar tanto a las peticiones que llegaron cuando ya el brazo había pasado por el lugar que ocupan.

Obsérvese que, en cualquiera de las dos variantes del elevador, Scan y C-Scan, el cabezal se mueve al final o al inicio del disco lo cual, a todas luces, es innecesario de ahí que las implementaciones de este algoritmo lo que en realidad hacen es mover el cabezal hasta la petición

más externa de la dirección en que van, sin arribar al final del disco. Esas variantes se conocen como Look y C-Look.

Selección del algoritmo de planificación

Entre los algoritmos SSTF y FCFS es más usual que se escoja el primero debido a que tiene mejor rendimiento que el segundo. En sistemas que tienen una carga pesada con relación al uso de los discos, es conveniente usar los algoritmos del elevador debido a que tienen menor probabilidad de provocar inanición.

Está claro que puede encontrarse un orden óptimo para una secuencia particular de peticiones al disco, pero eso no es válido en el mundo de la computación donde es preciso encontrar una generalidad y eso no resulta fácil porque la forma en que se comporte cada algoritmo dependerá de las secuencias de peticiones que existan, la cual no es predecible.

Por otra parte, el servicio que brinde el disco dependerá del método de asignación de bloques de datos que utilice el sistema de archivo, de la localización de los directorios y bloques de índices. El sistema de archivo es el tema del capítulo siguiente.

Los algoritmos analizados solo toman en cuenta el tiempo de acceso (seek time). Sin embargo, en los discos modernos el tiempo de rotación puede ser considerable, pero es muy difícil tomarlo en cuenta debido a que no se conocen las posiciones físicas de los bloques lógicos. Actualmente los fabricantes de discos incluyen algoritmos de planificación en el controlador de los discos, con lo cual se disminuyen estos problemas, en este caso el SO envía un lote de peticiones al controlador que se encarga de encolarla y planificarla tomando en cuenta los tiempos de acceso y de rotación.

Es usual que los algoritmos de planificación de disco se escriban como módulos separados del SO, lo cual facilita su reemplazamiento para ajustarlos a características específicas.

Debe tenerse en cuenta que los discos no solo se usan para E/S de archivos sino también para otras tareas; por ejemplo, como discos de resguardo para la memoria virtual. Lo anterior determina que, en muchos casos, sea necesaria la intervención del SO en los algoritmos de planifi-

cación; por ejemplo, debe privilegiarse una demanda de página sobre una petición de E/S de archivo.

3.2.2. El formato de los discos

Los discos duros reciben un formato a bajo nivel o físico que los divide en sectores sobre los que el controlador efectúa las operaciones de lectura o de escritura. Ese formato, regularmente, se hace en la misma fábrica; pero existen herramientas que les permiten a los usuarios dar formato a bajo nivel, lo cual puede resultar útil cuando los discos están dañados.

El formato a bajo nivel llena cada sector del disco con una estructura de datos especial que está formada por una cabeza una zona para los datos (usualmente 512 bytes) y una parte trasera.

La cabeza y la parte trasera se usan para almacenar información que puede leer el controlador del disco; por ejemplo, el número del sector y el código de corrección de errores, entre otros datos útiles.

Antes de usar un disco duro, que tiene formato físico, deberán realizarse dos labores:

- La primera consiste en dividir el disco en uno o más conjuntos de cilindros vecinos denominados particiones. El SO tratará a cada partición como un disco lógico y hoy en día es usual tener una partición para instalar el SO y otra para datos, aunque pueden ser más en dependencia del gusto de los usuarios, las necesidades reales y el tamaño del disco.
- El segundo paso es darle un formato lógico a cada partición. En este proceso la herramienta para formatear almacena las estructuras de datos iniciales del sistema de archivo; por ejemplo, mapas de bloques libres y asignados y también se asigna espacio para el directorio raíz, todo lo cual dependerá de la estructura del sistema de archivo que se use.

Carga (bootstrap) del SO y su relación con los discos

La carga del SO está antecedida por un conjunto de acciones que forman parte del firmware del equipo y se realiza cuando se enciende o reinicia una computadora, entre ellas pueden citarse las siguientes:

- Realizar un diagnóstico del *hardware* básico del equipo de cómputo y los demás equipos conectados. Este proceso se conoce como POST (Power On Self Test).
- Después del POST se carga en memoria el cargador del equipo (bootloader) y se le da el control.
- El bootloader busca el cargador del SO en un área especial de un equipo externo (conocida como boot), lo carga en la memoria y le da el control. La mayoría de las veces este cargador reside en discos duros, pero también puede estar sobre CD, DVD, memoria flash e incluso en un equipo remoto.

Cuando el cargador del SO toma el control comienza a traer a memoria el SO. Este proceso puede ser bastante complejo, es dependiente del SO y muchas veces incluye la carga de varios cargadores parciales que van trayendo a memoria distintas partes del SO. Entre los cargadores típicos se pueden citar los siguientes:

- GNU Grand Unified Bootloader (GRUB): gestor de arranque múltiple (capaz de cargar distintos SO) desarrollado como parte del proyecto GNU. Se utiliza, principalmente, cuando hay más de un SO instalado sobre un mismo equipo (sobre distintas particiones).
- NT Loader (NTLDR): cargador de las primeras versiones de Windows NT, incluyendo Windows XP y Windows Server 2003.
- BOOTMGR (Boot manager): gestor de arranque múltiple para los SO Windows 7, 8, 10 y Windows Vista.
- Linux Loader (LILO): gestor de arranque múltiple para los SO Linux y otras plataformas.
- Las particiones de los discos duros pueden manejarse por medio de dos técnicas:
- Master Boot Record (MBR): es la forma más antigua y por eso cualquier SO puede manejarla. Admite formatos de particiones primarias, extendidas y lógica y manipula, normalmente, hasta dos terabytes de espacio en disco.
- Guid Partition Table (GPT): este tipo de partición puede manejar discos de cualquier longitud, pero al momento de escribirse este libro aún no la admiten todos los SO.

Disco	Tipo	Sistema de archivo	Capacidad	Tipo de dispositivo	Estilo de partición	
Disco 0 Disco 1 Disco 2 Disco 3 CD ROM	Básico Básico Básico Extraíble DVD (D)	NTFS NTFS FAT 32 NTFS CDFS	931,39 GB 931,51 GB 931,51 GB 29,30 GB 428,00 MB	SATA USB USB SATA SATA	GPT MBR MBR MBR MBR	

Disco 0 Básico 931,39 GB	260 MB Partición de sistema EFI	Windows (C) 390,60 GB NTFS Partición primaria, arranque, archivo de paginado, volcado	Data (E) 509,07 GB NTFS Partición primaria	1000 MB Partición de recuperación	30 GB Partición de recuperación	500 MB Partición OEM
Disco 1 Básico 931,51 GB	Externo 1 (F) 931,51 GB NTFS Partición primaria					
Disco 2 Básico 931,51 GB	Externo 2 (G) 931,51 GB FAT 32 Partición primaria					
Disco 3 Extraíble 931,51 GB	MATEOLEZ (H) 29,30 GB NTFS Activo, Partición primaria					
CD ROM 0 428 MB CD de audio	Audio CD (D) 428 MB CDFS Partición primaria					

Figura 88. Sistema operativo Windows 10. Particiones.

La figura 88 muestra el estado de las particiones en un sistema al que están conectados tres discos duros (0, 1, 2), una memoria flash (disco 3) y un DVD de música. La figura ha sido editada a partir de una vista tomada con el administrador de discos del SO Windows 10.

- El disco 0, está particionado al estilo GPT y se conecta a la computadora a través de un puerto SATA (Serial Advanced Technology Attachment):

Su primera partición (260 MB) es de sistema EFI (Extensible Firmware Interface). Las particiones EFI se usan en computadoras que se adhieren a la interfaz UEFI (Unified Extensible Firmware Interface). Cuando se inicia una computadora de este tipo, el firmware UEFI carga varios archivos utilitarios y los archivos que permiten iniciar el SO.

La última partición la puso el fabricante OEM (Original Equipment Manufacturing) y las dos que le anteceden (1000 MB y 30 GB) son de recuperación.

El propietario de la computadora dividió el disco 0 en dos particiones

válidas para el trabajo, que serán las que verá el usuario, ambas usan el sistema de archivo NTFS (New Technology File System). La primera, denominada Windows, la dedicó a la instalación del SO Windows 10; por ese motivo es una partición de arranque lo que significa que en ella reside un cargador que es capaz de cargar ese SO. La segunda, denominada Data, está destinada a almacenar el trabajo del usuario.

- Los discos 1 y 2 usan el estilo de partición MBR y se dedican a datos, el disco 1 se formateó con el sistema de archivo NTFS, mientras el disco 2 se formateó con el sistema FAT32 (File Allocation Table) para destinarlo a intercambiar información con otros medios (ambos discos están conectados a puertos USB).
- El disco 3 es un medio extraíble (una memoria flash en este caso) que está formateado con el sistema de archivo NTFS.
- Por último, en la unidad de CD/DVD se ha insertado un CD de audio que está formateado con el sistema de archivo CDFS (Compact Disk File System).

El sistema de archivo es el contenido del capítulo siguiente y por eso, en este momento, no se ofrecen más detalles acerca de este aspecto.

La caché del disco

La memoria caché, en general, se refiere a una memoria que es menor y más rápida que la memoria principal; desde el punto de vista lógico está situada entre la memoria principal y el procesador con el objetivo de agilizar los accesos a la primera.

El caché de disco sigue los mismos principios que la caché de memoria. En este caso es un búfer, en memoria principal, que contiene una copia de algunos sectores del disco.

Cada vez que se realiza una operación de E/S a un disco determinado, se verifica si el sector solicitado está en su caché, si es así se accede a él directamente agilizando el proceso. En caso de que el sector no esté en el caché se lee desde el disco y se guarda una copia de él en la caché del disco, garantizando que un futuro acceso, al mismo sector, sea más rápido.

El principio de localidad establece que cuando se referencia un bloque de dato, en una operación simple de E/S, se puede esperar que el mismo dato sea referenciado nuevamente en un futuro cercano. La ca-

ché se basa en ese principio, queda claro que el búfer que se dedica a este trabajo es muchísimo más pequeño que la capacidad del disco, de ahí que cuando se llena ese búfer se sobrescribe sobre los datos viejos, para lo cual se utiliza algún algoritmo de reemplazamiento (LRU, LFU-Least Frequently Used, entre otros.).

3.3. Conjunto redundante de discos independientes

Un factor importante a tomar en cuenta con relación al almacenamiento de datos sobre discos simples es la baja velocidad de estos medios en relación con la memoria principal y el procesador. Para alcanzar una mayor eficiencia surgieron los RAID (Redundant Array of Independent Disks) que no es más que un conjunto de discos que actúan de manera independiente y en paralelo, con la característica adicional de que los mismos datos están almacenados en más de un disco del conjunto; lo que ofrece un nivel de redundancia que hace menos probable la pérdida de datos.

Los RAID pueden configurarse con discos duros o con discos SSD (Solid State Drive) y es posible implementarlos por *hardware* o por *software*. Existen siete niveles de RAID:

- RAID de nivel 0: también conocido como conjunto dividido, distribuye los datos equitativamente entre dos a más discos. En este caso no hay redundancia y por eso algunos autores no consideran este nivel como un miembro real de la familia RAID, aunque sí se logra mayor eficiencia cuando se realizan peticiones de E/S que están sobre distintos bloques de datos en discos diferentes, debido a que se pueden efectuar en paralelo.
- RAID de nivel 1: en este caso la redundancia se logra duplicando los datos o sea cada disco del conjunto tiene un disco espejo que contiene los mismos datos (por eso también se conoce como espejo), en este caso una petición de lectura puede satisfacerse por cualquiera de los discos que poseen el dato, una petición de escritura necesita que se actualicen ambos discos, pero esa tarea puede hacerse en paralelo.

Debe observarse que los datos seguirán estando accesibles cuando uno de los discos que los contiene falle, aunque esta redundancia tan alta se paga con el costo.

- RAID de nivel 2: los niveles 2 y 3 utilizan técnicas de acceso paralelas debido a que todos los discos del conjunto participan en cada petición de E/S, los datos están divididos en unidades pequeñas que pueden ser bytes o palabras, cada dato posee un código de corrección de error que se calcula a partir de los bits de cada disco que lo conforma, típicamente se usa código Hamming. RAID 2 es menos costoso que RAID 1 pero también tiene un precio elevado con relación a los restantes.
- RAID de nivel 3: su organización es similar a RAID 2, pero solo necesita un disco para la redundancia y emplea acceso paralelo sobre datos distribuidos en pequeñas unidades. El código de corrección de error se obtiene con un bit de paridad simple calculado a partir del conjunto de los bits individuales que están en la misma posición de todos los discos. Debido a que los datos están divididos en unidades pequeñas, distribuidas sobre los discos, se logran altas tasas de transferencias.
- RAID de nivel 4: en los niveles superiores (4 al 6) cada disco actúa de manera independiente y por eso son más adecuados para aplicaciones que necesitan grandes volúmenes de E/S pero menores tasas de transferencias. En estos esquemas (4 al 6) el tamaño en que se dividen los datos es relativamente grande.
- RAID de nivel 5: la diferencia con relación al nivel 4 es que el bit de paridad se distribuye a través de todos los discos lo que evita potenciales cuellos de botellas en las E/S. En este nivel la pérdida de uno de los discos no implica la pérdida de los datos.
- RAID de nivel 6: en este caso existen dos cálculos para el bit de paridad que se almacena sobre bloques de discos diferentes, lo que permite regenerar los datos cuando uno de los discos que los contiene falla.

Se pueden ofrecer muchos más detalles acerca de los RAID, pero un análisis más profundo de estos aspectos no se incluye entre los objetivos del texto.

3.4. Discos ópticos (CD y DVD)

Los CD y DVD son discos ópticos que permiten grabarse y leerse usando un rayo láser que codifica la información por medio de surcos microscópicos. El láser utiliza especificaciones diferentes para los CD y DVD en cuanto a la longitud de onda y otros parámetros.

Los discos ópticos se fabrican de policarbonato u otro material plástico

y se recubren de una capa, usualmente de aluminio, que refleja la luz del rayo láser en el espectro infrarrojo (no visible al ojo humano), a la cual se le superpone una capa protectora.

Los discos compactos (CD) pueden almacenar hasta 700 MB de datos, aunque inicialmente se concibieron para almacenar audio. Su popularidad como medio de almacenamiento comenzó a decaer desde el año 2000 pero aún se utilizan. Los DVD pueden grabar hasta 4,7 GB de datos.

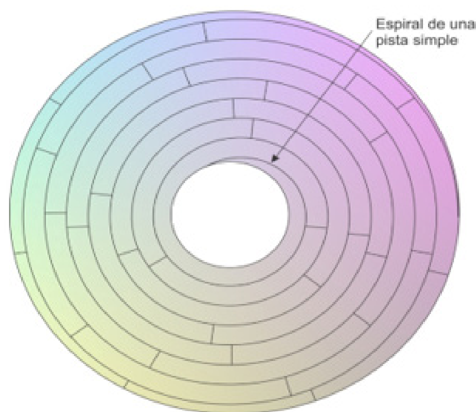


Figura 89. Vista de una pista de un disco óptico. Particiones.

El patrón de codificación se basa en pistas que siguen un recorrido en espiral continuo, extendido desde el centro del disco hacia afuera (figura 89). La parte interna del disco se lee a mayor velocidad que la externa.

Existen varios tipos de CD y de DVD:

Tipos de CD

- De solo lectura o CD-ROM (Compact Disk Read Only Memory).
- Grabable CD-R (Compact Disk Recordable): permiten varias sesiones de grabación, pero la información grabada en sesiones anteriores no puede sobrescribirse ni borrarse, solo pueden hacerse nuevas sesiones en los espacios libres. En realidad la facilidad de las sesiones no se usa prácticamente hoy en día.

- Regrabable CD-RW (Compact Disk Re-Writable): pueden grabarse muchas veces.

Tipos de DVD

- De solo lectura o DVD-ROM (*Digital Versatile Disk Read Only Memory*).
- Grabable DVD-R (*Digital Versatile Disk Recordable*): solo pueden grabarse una vez.
- Regrabable DVD-RW (*Digital Versatile Disk Re-Writable*): pueden grabarse muchas veces.

Los lectores de discos están compuestos, entre otros elementos, por las unidades siguientes:

- Un cabezal con un emisor de rayos laser para enviar su luz a la superficie del disco y una foto receptor que recibe el haz de luz que rebota en la superficie.
- Dos motores: uno para girar el disco y otro para mover el cabezal radialmente.
- Un convertidor de señal digital a analógica.

Los discos ópticos están siendo sustituidos por las memorias USB y todo parece indicar que les sucederá lo mismo que los discos flexibles (disquetes o floppy disk, que es un soporte magnético extraíble de almacenamiento de datos) de los cuales ya casi ni se habla. Hoy en día muchas computadoras se fabrican sin unidades lectoras de discos ópticos, tal y como comenzó a suceder hace años con las unidades lectoras de discos flexibles.

3.5. Memoria USB

Popularmente se le da el nombre de memoria USB (Universal Serie Bus que es un canal estándar para transferir datos) a los dispositivos de almacenamiento extraíbles que usan memoria flash, esta última se derivó de la memoria EEPROM (Electrically Erasable Programmable Read Only Memory) pero, a diferencia de ella, permite operaciones de lectura/escritura de múltiples posiciones en una sola operación.

Otros nombres comunes que se le dan a la memoria USB, son: lápiz de memoria, lápiz USB, pen drive.

Entre las ventajas de la memoria USB pueden destacarse las siguientes:

- Cumple la Ley de Moore. Desde su introducción en el mercado, han aumentado sus capacidades y disminuido sus precios.
- Tienen buena resistencia a los golpes, bajo consumo de energía y considerable velocidad.
- No poseen partes móviles y por eso no hacen ruidos.
- Su pequeño tamaño, ligereza y versatilidad la hacen adecuada para soporte móviles de información.

Este tipo de dispositivo también tiene las siguientes desventajas:

- Solo permite una cantidad limitada de escrituras y borrados. Aunque esta cantidad es considerablemente alta.
- La relación costo/capacidad es menos favorable con relación a los discos duros y ópticos.

Los puertos USB que se usan actualmente siguen las normas USB 2.0, 3.0 y 3.1 con velocidades máximas de 10 Gbit (un Gbit equivale a 109 bits), 5 Gbit y 480 Mbps (un Mbps equivale a 1000 kb/s) respectivamente. Las versiones más actuales manejan la energía más eficientemente que sus predecesores. USB 3.0 y 3.1 pueden enviar y recibir datos a la misma vez, mientras USB 2.0 solo recibe o envía en cada momento. Se puede conectar una memoria USB a un puerto con norma superior pero la velocidad de transferencia de datos estará limitada por ese puerto.

Las memorias USB admiten que se le instale un cargador de algún SO de manera que si la BIOS lo permite puede iniciarse el SO desde ella, lo cual resulta muy conveniente para los procesos de instalación y reparación de estos sistemas. Esta capacidad la utilizan varios SO que se distribuyen completos sobre este tipo de memoria.

3.6. Unidades de estado sólido

Una unidad de estado sólido, más conocido como SSD por sus siglas en inglés, es un dispositivo que utiliza memoria no volátil como medio de almacenamiento. Este tipo de disco no tiene partes móviles y por eso es menos sensible a los golpes y prácticamente no produce ruidos. Los SSD ofrecen mejores tiempos de respuesta, pero su vida útil es muy inferior a la de los discos tradicionales.

Actualmente la mayoría de los SSD utilizan memoria flash basada en puertas NAND (Not AND, puerta lógica que realiza la operación producto lógico negado), lo que permite retener los datos sin alimentación

eléctrica, aunque también se construyen con memoria de acceso aleatorio.

También se han desarrollado dispositivos que combinan discos duros con memorias flash, denominados unidades de estado sólido híbridas (SSHD), el objetivo es conseguir mayor capacidad y velocidad que los discos tradicionales a precios inferiores a los SSD.

Debe observarse que no son aplicables todos los algoritmos de planificación analizados en el epígrafe 3.1.1 a los SSD (ni a la memoria USB), debido a que no existen los movimientos mecánicos que caracterizan los discos magnéticos, ni tampoco los platos, cilindros, entre otros.

Resumen del capítulo

Actualmente los discos magnéticos constituyen el medio de almacenamiento principal para la mayoría de las computadoras.

En un futuro, los discos de estados sólidos pudieran ser los sustitutos de los discos duros, debido a que los primeros son mucho más rápidos, pero no se avizora que eso ocurra dentro de poco tiempo debido a que los SSD son menos duraderos, mucho más caros y de menor capacidad que los discos duros. Por ese motivo se han creado los SSHD que utilizan ambas tecnologías para tomar lo mejor de cada una abaratando costos, obteniendo velocidades intermedias y dando más duración a los nuevos equipos.

Las peticiones hechas a los dispositivos externos se generan desde el sistema de archivo o desde el sistema de memoria virtual, cada petición especifica su dirección en el dispositivo por medio de un número que hace referencia a una unidad lógica.

Para satisfacer una petición de E/S sobre un disco duro se pueden usar diferentes algoritmos que deben tener en cuenta los tiempos de acceso (seek time) y de latencia (latency time), pero no es posible determinar la eficiencia de ellos en forma general porque normalmente los tiempos de acceso estarán en dependencia de la secuencia de peticiones. Los algoritmos analizados son: FCFS, SSTF, Scan, C-Scan, Look y C-Look.

Debido a que los SSD y las memorias USB no tienen movimientos mecánicos es habitual usar el algoritmo FCFS para satisfacer las peticiones de E/S hechas a esos dispositivos.

La tecnología RAID ofrece mejores rendimientos que los discos individuales y también mayor tolerancia a fallos, sus costos se están reduciendo continuamente y parece ser la variante del futuro. Existen siete niveles de organización de los RAID (0 a 6), cada uno de ellos tiene ventajas y desventajas que deben tomarse en cuenta en el momento que deseen utilizarse.

Este capítulo es una lectura obligada para entender el siguiente debido a que el sistema de archivo se basa en el funcionamiento de los equipos de almacenamiento masivos.

3.7. Ejercicios propuestos

1. **Explique los siguientes algoritmos de planificación de disco:**

a) FCFS.

b) SSTF.

2. **Explique por qué SSTF tiende a favorecer las peticiones que están en los cilindros de la parte media del disco.**
3. **Explique los algoritmos de planificación de disco Scan y C-Scan. Establezca las diferencias entre ellos.**
4. **Establezca las diferencias entre los algoritmos de planificación de disco Scan y Look ¿Cuál de ellos utilizaría? ¿Por qué?**
5. **Establezca las diferencias entre los algoritmos de planificación de disco C-Scan y C-Look ¿Cuál de ellos utilizaría? ¿Por qué?**
6. **¿Por qué se dice que entre los algoritmos de planificación de discos analizados el único candidato para planificar unidades SDD es el FCFS?**
7. **Haga una comparación entre los niveles RAID estableciendo semejanzas y diferencias entre ellos.**
8. **Presente una secuencia de peticiones a discos y dibuje un gráfico semejante que permita analizar el comportamiento de los algoritmos analizados en este capítulo.**

a) Haga un análisis crítico del compartimento de cada algoritmo y de la eficiencia lograda por la secuencia que usted ha elegido.

b) ¿Cuál resultó ser mejor en este caso? ¿pueden generalizarse conclusiones a partir de este análisis?

Capítulo IV. Sistema de archivos

4.1. El sistema de archivos y sus características

El sistema de archivo es el responsable de controlar la memoria externa que está soportada sobre diferentes equipos, algunos de los cuales se analizaron en el capítulo III.

El trabajo más común de cualquier usuario es interactuar con el sistema de archivo; guardando, extrayendo o procesando información que está almacenada en archivos que residen sobre diversos soportes externos, para lo cual utilizan algún programa específico; debido a eso se dice que esta es la parte más visible de cualquier SO.

El sistema de archivos proporciona una abstracción a los usuarios que los libera de la necesidad de conocer la forma en que se guarda la información sobre los equipos de memoria secundaria, permitiéndoles hacer diversas acciones sobre los archivos como pueden ser: crearlos, eliminarlos, copiarlos, moverlos, entre otros., de una manera cómoda, homogénea y sencilla.

Como su nombre lo indica, el sistema de archivo basa todo su trabajo en el concepto abstracto de archivo.

4.2. Los archivos

Los archivos proporcionan una forma lógica y única para referirse a la información que se almacena sobre los diferentes equipos de almacenamiento, sin importar la naturaleza física de cada uno de ellos.

El SO mapea cada archivo sobre algún equipo físico que mantiene la información en forma permanente, o sea persiste cuando se apague o reinicie la computadora.

Conceptualmente un archivo es una colección de información relacionada que responde a un nombre y generalmente está almacenada en algún equipo de almacenamiento externo. Se dice generalmente porque muchos SO tratan a algunos periféricos de E/S, que no son de almacenamiento, como si fueran archivos; por ejemplo, una impresora.

Todo archivo tiene una estructura básica definida por su creador que

puede estar formada por: bits, bytes, líneas, registros, entre otros. El archivo en sí es una secuencia de esa estructura básica.

Como puede apreciarse el concepto de archivo es muy general, lo que permite almacenar información con diferentes formatos y cada uno de ellos deberá manipularse con programas específicos, por ejemplo: un archivo texto para un editor dado es una secuencia de caracteres organizada en líneas y páginas, un programa ejecutable está formado por un conjunto de secciones de código binario que el cargador del SO puede traer a memoria para ejecutarlo.

4.3. La arquitectura del sistema de archivos

Para comprender la funcionalidad del sistema de archivo resulta adecuado analizar la figura 90, que muestra una visión de la organización de esta parte del *software*, la estructura real variará de un sistema de archivo a otro.

Figura 90. Arquitectura típica del sistema de archivo.

En el nivel inferior, que se denominará Controlador de entrada salida, residen los manipuladores de equipos (device drivers), los cuales tiene la responsabilidad de transferir información entre equipos de diversos tipos y la memoria. Estos manipuladores se comunican directamente con los periféricos (E1…En), sus controladores o sus canales.

Los manipuladores de equipos tienen la responsabilidad de iniciar las operaciones de E/S sobre un equipo dado hasta completar la solicitud recibida, ellos tienen que lidiar con las características específicas de cada medio.

Por encima del primer nivel se sitúa el Sistema de archivo básico que tiene la responsabilidad de servir como una interfaz entre sus dos vecinos

(superior e inferior). Para el caso de los equipos que organizan la información en bloques de datos, como los discos y las cintas, intercambia los bloques entre los equipos y la memoria, es decir tiene la responsabilidad de poner un bloque dado sobre el equipo de almacenamiento o de situar el bloque en la memoria principal. Este nivel no entiende el contenido del dato ni la estructura del archivo que lo contiene.

Después se sitúa el Supervisor básico de E/S que es responsable de todas las inicializaciones y terminaciones de E/S sobre archivos, para realizar su trabajo mantiene varias estructuras de control que gestionan el equipo de E/S, planificándolo y conservando el estatus del archivo. Aunque muchos autores consideran a las partes precedentes como componentes del SO (y otros no), el supervisor básico de E/S es, sin lugar a dudas, parte del SO.

El nivel de Entrada/salida (E/S) lógica trata con unidades de información agrupadas en registros, o sea donde el nivel precedente ve bloques de datos él ve archivos formados por registros que tienen un formato dado. Este nivel es el más cercano a los usuarios y provee una interfaz estándar entre las aplicaciones, el sistema de archivo y los equipos donde están los datos.

Cada registro (record) está formado por una colección de campos (field) relacionados, siendo este último el elemento de dato básico. Los campos se caracterizan por su tipo (cadena, entero decimal, entre otros.) y su longitud. Tanto los campos como los registros pueden ser de longitud fija o variable.

4.4. Directorios

Generalmente los directorios, son archivos especiales que contiene información acerca de otros archivos y directorios. Se dice generalmente porque el directorio raíz no es un archivo y se construye cuando se formatea lógicamente la unidad de almacenamiento.

La función principal de los directorios es localizar archivos, pero también pueden mantener campos adicionales para brindar información acerca de ellos, por ejemplo: el tipo de archivo, datos de protección, la identificación de su propietario, entre otros.

El conjunto de información asociado a un archivo se conoce con el

nombre de atributos del archivo.

La figura 91 ofrece una vista general de un directorio hipotético. Cada columna se ha identificado por un texto para facilitar la explicación, pero debe quedar claro que esa entrada no existe en la tabla de directorio.

Nombre	Tipo	Longitud	Fecha	Propietario	Permisos	Localización
nominaJunio	texto	12456	30-6-2018	mmadiedo	rw	123
nominas	ejecutable	2345	11-3-2015	mlezcano	x	312
---	---	---	---	---	---	---

Figura 91. Tabla de directorio hipotética.

El directorio ejemplificado en la figura 91 contiene dos archivos:

- El primero, denominado nominaJunio, es un archivo texto de 12456 bytes que fue creado su propietario (mmadiedo) el día 30-6-2018. Sobre el archivo se pueden efectuar las operaciones de lectura y escritura (rw).
- El segundo archivo se nombra nóminas y es un ejecutable.

El último campo de cada entrada en la tabla de directorio, nombrado localización en la figura, le indica al SO donde comienza cada archivo, lo cual dependerá del tipo de sistema de archivo que se use.

Las restantes entradas del directorio de la figura 91 (sus filas), están vacías y por eso admitirán referencias futuras a otros archivos.

La manera en que se estructura la información de los directorios depende del sistema de archivo, la forma más simple es mantener una secuencia de entradas, una por cada archivo, tal y como se aprecia en la figura 91.

Sobre un directorio pueden realizarse las siguientes operaciones generales:

- Buscar un archivo o directorio.
- Crear un archivo o directorio nuevo.
- Borrar un archivo o directorio.
- Listar el directorio.
- Actualizar el directorio.

De acuerdo a las facilidades brindadas por cada sistema de archivo se podrán realizar otras operaciones más específicas, o se podrán agre-

gar opciones a las listadas anteriormente para hacerlas más particulares; por ejemplo, la acción de listar el directorio puede incluir alguna opción para que solo se listen los archivos de un tipo (ejecutables, textos, entre otros.).

La figura 92 muestra un ejemplo de la estructura jerárquica de muchos directorios actuales.

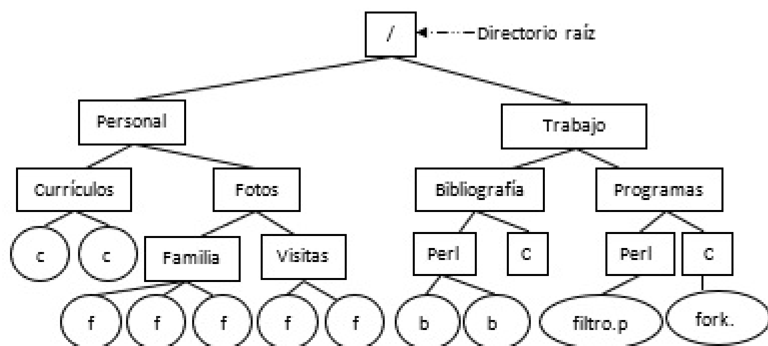


Figura 92. Estructura jerárquica de un directorio. hipotética.

Un directorio puede contener archivos y otros directorios, representados por elipses y rectángulos respectivamente en la figura y existe un directorio especial denominado raíz (identificado por / en la figura).

Para encontrar la ubicación de un archivo dentro de esa jerarquía se sigue la ruta o camino (path) hasta él. Dichas rutas pueden ser relativas o absolutas.

Las rutas absolutas parten del directorio raíz, por ejemplo: /Personal/Fotos/Familia/f1 y /Personal/Fotos/Visitas/f1 identifican dos archivos que se nombran igual (f1) pero están en directorios distintos.

Las rutas relativas parten del directorio actual o de trabajo, entendido como tal aquel en el que se está trabajando (el usuario o un programa se cambió hacia él), por ejemplo:

- Si el directorio actual es /Trabajo/Bibliografía/Perl se podrán referir todos los archivos de ese directorio (b1 y b2) sin antecederle ningún camino.
- Si el directorio actual es Bibliografía, los archivos b1 y b2 se pueden referir usando el camino relativo Perl/b1 y Perl/b2 respectivamente.

Es bueno aclarar que esta es una forma que adopta el SO para organizar los archivos, pero en realidad no existen esas divisiones en el soporte de almacenamiento; por ejemplo, en los discos magnéticos solo existen las pistas y los sectores (capítulo III).

El SO debe brindar herramientas para mantener la estructura del directorio y navegar por ella, facilitando acciones tales como:

- Crear directorios de diversos tipos.
- Cambiar de un directorio a otro.
- Eliminar directorios, entre otros.

4.5. Compartir archivos

Los sistemas operativos multiusuario permiten que muchos usuarios estén trabajando sobre el mismo equipo de cómputo, en ese entorno resulta natural la necesidad de compartir archivos y realizar trabajos colaborativos. Para compartir archivos deben analizarse dos aspectos importantes que son los derechos de acceso y la posibilidad de accesos simultáneos.

Es habitual que los derechos de acceso sobre algún archivo o directorio se agrupen por usuarios o grupos de usuarios, los permisos pueden ser:

- Ninguno. No se conoce la existencia del archivo y por eso no se puede listar el directorio que lo contiene.
- Conocer. Se conoce la existencia del archivo o directorio y la identificación de su propietario. En este caso puede solicitársele permiso al propietario para hacer otras acciones.
- Ejecución. Puede cargarse y ejecutarse el programa contenido en el archivo. No puede copiarse.
- Lectura. Puede leerse el archivo o directorio.
- Agregar. Pueden adicionarse datos al archivo, típicamente al final, pero no modificar o eliminar su contenido.
- Actualizar. Pueden borrarse y modificarse datos del archivo.
- Modificar permisos. Pueden cambiarse los permisos de acceso, típicamente solo el propietario del archivo o algún administrador tiene este derecho.
- Borrar. Puede eliminarse el archivo o directorio.

El acceso simultáneo a un archivo puede traer inconsistencias, por eso

deben existir políticas de manejo para evitar ese problema. Queda claro que la lectura simultánea sobre un mismo archivo puede permitirse sin peligro alguno, pero si varios usuarios tienen permiso de agregar o actualizar pueden ocurrir inconsistencias en el contenido del archivo.

Una solución drástica para evitar las inconsistencias es bloquear todo el archivo cuando se está actualizando, pero una más específica sería bloquear solo los registros (records) involucrados en la actualización.

4.6. Métodos de asignación

Cuando se crea un archivo cualquiera es necesario asignarle espacio en memoria secundaria (un disco duro, una memoria usb, un disco de estado sólido, un cd o dvd, entre otros.), para lidiar con este problema se pueden seguir dos alternativas:

- La primera es reservar todo el espacio que necesitará el archivo.

Esta alternativa se conoce como pre asignación. En este caso es necesario conocer, a priori, la longitud que tendrá el archivo creado; lo cual es imposible la mayoría de las veces y por eso los usuarios deberán hacer un estimado, lo que puede causar que se sobreestime el espacio necesario (malgastándolo) o que sea insuficiente y por eso el archivo no podrá crecer todo lo que se necesita.

- La segunda es asignarle una pequeña unidad para después ir adicionándole más espacio de acuerdo a sus necesidades.

Esta manera de actuar se denomina asignación dinámica, en este caso el archivo se crea con un pequeño espacio para después ir solicitando más espacio cuando se necesite.

Otra consideración a tomar en cuenta es determinar el tamaño de las unidades de espacio que se soliciten, pudiera pensarse en establecer en un byte esa unidad; pero entonces sería necesario controlar cada byte del equipo de almacenamiento, para conocer cuándo están libres o ocupados. Esa forma de actuar resulta, a todas luces, inapropiada debido a que sería necesario usar una gran parte del equipo solo para llevar ese control.

Basado en el análisis anterior surgió el concepto de bloque que se refiere a la unidad minina de asignación de espacio sobre un soporte de almacenamiento, usualmente varios sectores.

Entonces es importante establecer un tamaño apropiado para los bloques: uno muy pequeño exigirá una estructura de datos muy grande para controlarlos (saber cuáles están ocupados y cuáles libres); uno muy grande malgasta espacio en el último bloque de los archivos que puede quedar prácticamente vacío. Tomando en cuenta estos aspectos debe tomarse una posición intermedia.

Como los bloques tienen una longitud dada, cuando se solicita espacio se asigna esa cantidad de bytes, que puede ser mayor que la pedida, perdiendo algunos bytes, a veces todos menos uno, en el último bloque del archivo. Este fenómeno se conoce como fragmentación interna y no tiene solución (Lezcano Brito, 2018).

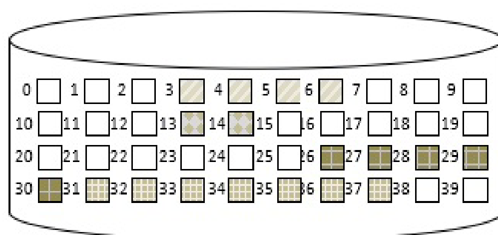
4.6.1. Asignación contigua

Cuando se crea un archivo con este tipo de política se asigna un conjunto de bloques al archivo nuevo, o sea se sigue una estrategia de pre asignación. Los bloques asignados tienen que estar contiguos es decir uno detrás del otro en el equipo de almacenamiento. El directorio deberá tener una entrada por cada archivo, en la cual se especifica el número que identifica al bloque de inicio y la longitud del archivo.

La figura 93 muestra el ejemplo de una unidad de almacenamiento en la cual residen cuatro archivos: Arch1, comienza en el bloque 3 y tiene 4 bloques; Arch2, inicia en el bloque 31 y tiene 7 bloques; Arch3 que tiene 2 bloques siendo el 13 el primero; Arch4 con bloque inicial 26 y 5 bloques de longitud. Los demás bloques del medio físico de almacenamiento están libres y tendrá que existir alguna estructura de datos para controlarlos.

Nombre del archivo	Bloque inicial	Tamaño
Arch1	3	4
Arch2	31	7
Arch3	13	2
Arch4	26	5

Directorio



Unidad de almacenamiento

Figura 93. Asignación contigua.

Esta manera de asignación tiende a formar conjuntos de bloques libres,

de diferentes longitudes, separados entre sí. Se le denominará hueco a cada uno de esos conjuntos.

Entonces, ante una petición de tamaño m deberá escogerse cuál de los huecos posibles se asignará. Existen tres estrategias para tomar la decisión respecto a cómo escoger ese espacio (Lezcano Brito, 2018):

- El mejor acceso: significa tomar un espacio que satisfaga la demanda y que sea el menor de todos los posibles.

Esta estrategia tiende a dejar espacios pequeños que podrían ser insuficientes para cualquier solicitud futura, de forma que el mejor acceso puede resultar la peor solución.

- El peor acceso: significa tomar un espacio que satisfaga la demanda y que sea el mayor de todos los posibles.

En este caso los espacios que van quedando son mayores, en general, y por eso tendrán mejores posibilidades de ser útiles.

- El primer acceso: significa tomar el primer espacio que satisfaga la demanda.

No se puede decir nada con relación a los espacios que deja esta estrategia debido a que sus resultados son impredecibles.

La asignación contigua tiene las siguientes ventajas (Lezcano Brito, 2018):

- Debido a que toda la información está contigua, puede considerarse la mejor para realizar accesos secuenciales dentro de un mismo archivo.
- Permite el acceso directo a los datos, para lo cual solo se necesita la dirección de inicio del archivo (está contenida en la tabla de directorio) y el desplazamiento a partir de esa dirección; por ejemplo, para acceder al cuarto bloque del archivo Arch2 mostrado en la figura 85, se toma la dirección de inicio del archivo (31) y se le suma 4, lo que permitirá acceder de forma directa al bloque 35.
- Es más rápido el acceso en los discos, debido a que el cabezal de lectura/escritura efectúa menos movimientos mecánicos para acceder a los datos que están unos a continuación de otros.
- La recuperación de datos perdidos en algún borrado es más fácil y hay mayores garantías de que sea exitosa.

Como casi todo en el mundo de la informática, esta forma de proceder también tiene algunas desventajas que son las siguientes (Lezcano Bri-

to, 2018):

- Un archivo solo puede crecer hasta el inicio de su vecino debido a que cualquier otro espacio libre no estará contiguo a él. Este problema hace que los usuarios tiendan a sobrestimar la longitud de los archivos y reserven espacios que en realidad son mayores que los que necesitan, malgastando ese recurso.

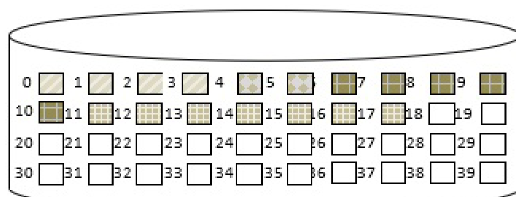
En la figura 85 se puede observar que el archivo arch4 no podrá crecer debido a que está limitado por arch2. Sin embargo, hay bloques libres que bajo otra estrategia podrían utilizarse.

- Provoca fragmentación externa, debido a que puede que no sea posible crear un archivo nuevo; aun cuando existan espacios no contiguos que sumados satisfacen la petición. En la figura 84, el mayor archivo que se puede crear es de 11 bloques debido a que ese es el mayor espacio libre contiguo que existe en ese momento (situado a partir del bloque 15). Sin embargo, hay un total de 22 bloques libres.

La mayoría de los sistemas de archivos que usan asignación contigua proveen algún mecanismo de desfragmentación para agrupar todos los espacios en uno solo, pero para hacer esta tarea es necesario detener todos los trabajos que se realizan sobre el periférico, lo cual puede resultar costoso.

Nombre del archivo	Bloque inicial	Tamaño
Arch1	0	4
Arch3	4	2
Arch4	6	5
Arch2	11	7

Directorio



Unidad de almacenamiento

Figura 94. Vista de la figura 85 después de desfragmentar.

La figura 94 muestra el efecto de desfragmentar el ejemplo de la figura 93. En este caso se ha dejado solo un espacio al final del equipo que está formado por 22 bloques libres, la misma cantidad que había en la figura 93 pero ahora están agrupados, lo que permitirá que el tamaño de un archivo nuevo solo dependa del espacio que existe y no de la fragmentación externa.

La acción de desfragmentar no tiene que dejar los archivos en el mismo orden que se muestra en la figura 94, ese detalle dependerá del algoritmo que siga el proceso de desfragmentación y, en general, no tiene mucho significado.

4.6.2. Asignación enlazada

La asignación enlazada se traza la meta de utilizar cualquier bloque libre, independientemente de si está o no consecutivo al anterior, con ese propósito los bloques asignados a un archivo se dividen en dos campos:

- El primero contiene el dato en sí.
- El segundo se usa como referencia al próximo bloque del archivo o sea contiene el número del próximo bloque. La figura 95 muestra la idea.

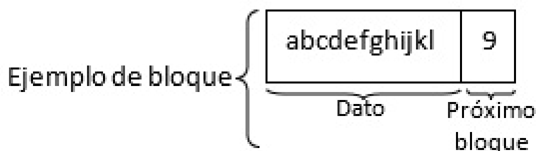


Figura 95. Bloque típico de asignación enlazada.

En este tipo de política de asignación, los bloques de un archivo tienden a estar dispersos sobre el equipo de almacenamiento y se elimina la fragmentación externa debido a que los bloques se pueden usar sin importar el lugar en que estén, la mala noticia es que para acceder a un bloque dado es necesario recorrer los anteriores, es decir no se permite el acceso directo.

Al igual que la asignación contigua, la tabla de directorio debe contener un campo para indicar el primer bloque del archivo y puede contener otro campo para especificar la longitud del archivo, esto último no es estrictamente necesario debido a que el último bloque del archivo podría contener un número especial en el área del puntero para indicar que es el último.

Nombre del archivo	Bloque inicial	Tamaño
Arch1	12	6

Directorio

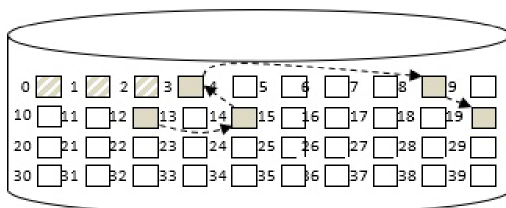


Figura 96. Asignación enlazada.

El archivo Arch1 de la figura 96 tiene 6 bloques y comienza en el 12, la cadena de bloques asignados al archivo es: 12, 14, 3, 8, 19; obsérvese que los bloques de este archivo están dispersos, pero pudieran estar contiguos total o parcialmente lo que dependerá del estado de los bloques ocupados y libres en el momento de la asignación. El último bloque, el 19 en este caso, podría tener un número especial en el área del puntero para indicar que es el último.

La dispersión de los bloques hace que el acceso sea más lento, en general, que en la asignación contigua y que no sea posible aplicar políticas de localidad, debido a que cuando se necesiten varios bloques será necesario recorrerlos secuencialmente. Otra desventaja es que se gasta espacio en el campo dedicado al puntero, donde no se guardan datos útiles desde el punto de vista del usuario (Lezcano Brito, 2018).

La principal desventaja de esta forma de asignación es que no permite el acceso directo debido que a que para acceder a un bloque dado será necesario recorrer los que le anteceden.

4.6.3. Asignación indexada

En la asignación indexada cada archivo necesita un bloque extra para almacenar índices hacia sus restantes bloques (de ahí el nombre). Cada entrada en la tabla de directorio apunta a un bloque de índices que indexa los siguientes bloques. En este caso (al igual que en la asignación enlazada), no es necesario conocer la longitud del archivo para establecer dónde termina (aunque sí es bueno que esa información esté disponible para otros fines) (Lezcano Brito, 2018).

La figura 97 esquematiza la idea, obsérvese que desde la entrada correspondiente al archivo Arch1 en la tabla de directorio se apunta a un

bloque (en este caso el 29) que se destina indexar los demás bloques del archivo (39, 2, 14, 15, 6).

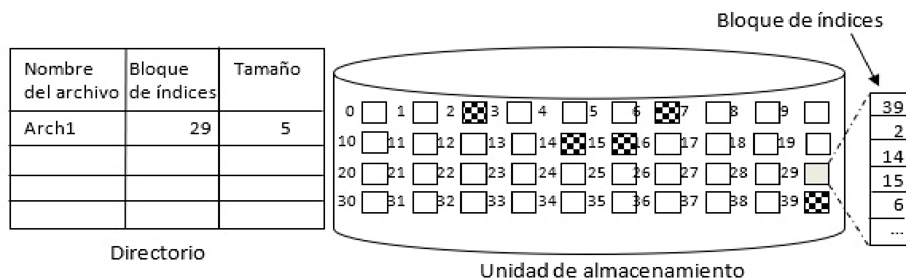


Figura 97. Asignación indexada.

La asignación indexada es la más utilizada actualmente y se le atribuyen las ventajas siguientes:

- No provoca fragmentación externa.
- Permite el acceso aleatorio.
- Los archivos pueden crecer mientras haya espacio y forma de hacer referencia a sus bloques (Lezcano Brito, 2018).

La principal dificultad se asocia al hecho de que los archivos tienden a estar dispersos, lo que puede incrementar el tiempo de acceso, pero pueden desfragmentarse las unidades cada cierto tiempo para minimizar estos efectos.

4.7. Manejo de los espacios libres

Sin importar el método de asignación, tiene que existir alguna manera de conocer los espacios que están libres y ocupados. Existen varias técnicas con ese propósito, algunas de ellas se detallan a continuación.

Mapas de bits

El método se apoya en un vector que tiene una longitud igual a la cantidad de bloques que existen en la unidad, si una entrada dada tiene el valor 0, el bloque al que se refiere está libre y si tiene un 1 está ocupado.

0	1	2	3	4	5	6	7	8	9	10
0	0	1	1	1	0	1	1	0	0	0

Figura 98. Mapa de bits.

En el ejemplo de la figura 98 se tiene una unidad de 11 bloques con la siguiente configuración en ese momento: los bloques: 0, 1, 5, 8, 9 y 10 están libres, mientras los bloques: 2, 3, 4, 6 y 7 están ocupados.

Hay que tomar en cuenta la búsqueda profunda dentro del mapa, ya puede ser muy lenta, sobre todo si la unidad está casi llena, para paliar esa situación muchos sistemas utilizan mapas de bits auxiliares que mantienen sub rangos de bloques libres con información acerca de la cantidad total de bloques libres, el mayor espacio consecutivo de bloques libres y otras informaciones que pudieran ser relevantes para agilizar los accesos.

Tabla de espacios libres

El espacio libre se puede controlar mediante una tabla con dos campos: el primero contiene la dirección de cada espacio libre y el segundo contiene su tamaño. El sistema de archivo toma de esa tabla el espacio que necesita para lo cual puede usar las estrategias del peor, mejor o primer acceso.

Una vez que se tome el espacio deberá actualizarse la tabla, para lo cual es necesario hacer las siguientes acciones:

- Si el espacio se ocupó totalmente, se elimina esa entrada de la tabla.
- Si solo se tomó parte del espacio, se restablecen los campos Inicio y Tamaño.

Este esquema tiene alguna sobrecarga dado que cada vez que se libere un espacio no basta con ponerlo en la tabla, sino que habrá que verificar si existía algún espacio libre antes y después de él, para ajustar la dirección de inicio y el tamaño.

Inicio	Tamaño
20	35
90	10
200	60
1024	100

Figura 99. Tabla de espacios libres.

La figura 99 muestra un ejemplo de una tabla de espacios libres, los es-

pacios podrían estar expresados en bytes, en ese caso será necesario dividir el tamaño del espacio libre entre el tamaño del bloque para conocer cuántos bloques se referencian desde cada entrada en la tabla.

Lista de bloques libres

Este método asigna un número secuencial a cada bloque y la lista de todos los bloques libres se mantiene en una parte reservada del equipo (Lezcano Brito, 2018) de almacenamiento. El espacio destinado para cada bloque dependerá de la capacidad del equipo debido a que es necesario referirse a todos sus bloques, pero ese espacio no es significativo y solo representa el 1% del espacio total del equipo.

4.8. Sistemas de archivo de los SO tipo Unix

Desde su surgimiento, los sistemas de archivos de los SO tipo UNIX han mantenido dos estructuras de datos sin cambios significativos, al menos con relación a sus objetivos y a los contenidos básicos, ellas son:

- Los nodos i, que contienen metadatos que son datos que describen otros datos (incluidos punteros a los datos reales).
- Los directorios, que se usan para localizar los archivos. En estos sistemas el directorio tiene solo dos entradas, para cada archivo:
  - La primera contiene el nombre del archivo.
  - La segunda contiene un puntero o referencia un bloque especial denominado nodo i. El nodo i contiene los demás datos del archivo, lo que incluye punteros a los bloques de datos.

Se puede apreciar que este sistema de archivo usa una política de asignación indexada donde cada archivo tiene un bloque de índice que es el nodo i.

4.8.1. Antecedentes y generalidades

Las primeras versiones de UNIX usaban un sistema de archivo nombrado FS (File System).

FS tenía una estructura muy simple que resultaba adecuada para los pequeños discos de la época e incluía los siguientes componentes (figura 92):

- Un bloque de carga del SO (booteo).
- Un superbloque para almacenar información acerca del disco.

- Una zona dedicada a los bloques para nodo i.
- Una zona dedicada a los bloques para datos.

Bloque de carga
Superbloque
Zona para bloques de nodos i
Zona para bloques de datos

Figura 100. Estructura de un disco con formato FS.

Cuando los discos comenzaron a hacerse mayores, la firma *Berkely Sostware Distribution* (BSD) le hizo mejoras con el fin de lograr accesos más rápidos y reducir la fragmentación, surgiendo el sistema de archivo Fast File System (FSS). FSS dividió los discos en grupos de cilindros. Al inicio de cada grupo incluyó una estructura de datos para controlar el espacio de los cilindros del grupo.

Los SO tipo Unix modernos soportan múltiples sistema de archivos pero en general se basan en los nodos de índices o nodos i (figura 101).

Propietario
Marca de tiempo
Permisos
... otros campos...
Punteros directos
Puntero simple
Puntero doble
Puntero triple

Figura 101. Nodo i.

Un nodo i es una estructura de datos que se almacena dentro de un

bloque y mantiene información acerca de un archivo dado (contiene metadatos). Existe un nodo i por cada archivo y todos están apuntados desde una tabla de directorio.

El nodo i especifica los permisos sobre el archivo, el identificador de su propietario, la hora en que se creó, su longitud, un conjunto de apuntadores a bloques, entre otros. Esos campos varían entre los distintos SO tipo UNIX.

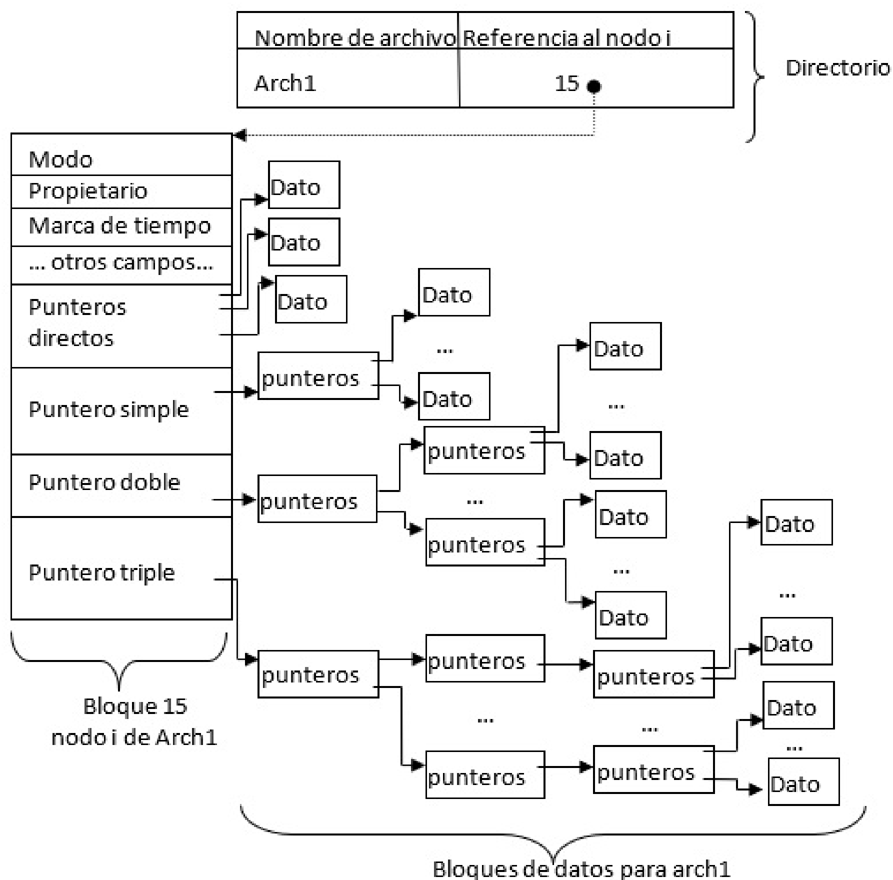


Figura 102. Relación entre la tabla de directorio y el nodo i.

Existen tres tipos de punteros dentro de un nodo i, que se usan para indexar los bloques de datos de cada archivo (figuras 101 y 102):

- El campo punteros directos es un espacio que contiene varios punteros a los primeros bloques del archivo. La cantidad de punteros varía de un UNIX a otro.
- Si los punteros anteriores resultaran insuficientes para referir los bloques de un archivo dado, se utiliza el campo puntero simple que apuntará a un bloque de índices (sería un metadato) que contiene punteros a bloques de datos.
- El campo puntero doble apunta a un bloque de índices que contiene punteros a bloques de punteros, los cuales apuntan a bloques de datos (una doble indirección). Este campo se usa cuando no es suficiente el anterior.
- El campo puntero triple es un puntero que apunta a un bloque de índices que contiene punteros a bloques de punteros, los cuales apuntan a bloques de índices que apuntan a bloques de datos (una triple indirección). El campo se usa cuando no alcanza el anterior.

En la figura 102, se ejemplifica la relación entre los directorios y los nodos i. En este ejemplo el directorio solo contiene el archivo Arch1 y el campo referencia al nodo i especifica que el nodo i de Arch1 es el bloque 15 que contiene toda la información restante del archivo (incluida su localización).

Muchos sistemas tipo Unix soportan apuntadores para archivos de 64-bit lo que da la posibilidad de que los archivos puedan tener varios exabytes (EB) de longitud.

Existen seis tipos de archivos en UNIX:

1. Regulares: pueden ser archivos de textos o de programas en general, el sistema de archivo los trata como torrentes de bytes.

2. Directorios: son archivos especiales que contienen una tabla de directorio, como la mostrada en la figura 19 (con el nombre de cada archivo y su respectiva referencia al nodo i). Se organizan en forma jerárquica y sobre ellos solo el sistema de archivo puede escribir y los programas de usuarios pueden leer.

3. Especiales: este tipo de archivo no contiene datos, se usan para proporcionar un mecanismo uniforme para mapear los nombres de los archivos especiales sobre equipos físicos. En UNIX cada equipo físico está asociado con un archivo especial, que lo maneja el sistema de

archivo cuando necesita leer o escribir en él; los equipos incluyen: manipuladores de discos y de cintas, terminales, líneas de comunicación, impresoras, entre otros. Este tipo de archivo reside, típicamente, en el directorio /dev.

4. Tuberías nombradas: las tuberías son mecanismos para comunicar dos procesos a través de un búfer sobre el que un proceso escribe y el otro lee. También existen las tuberías sin nombres que son conexiones temporales.
5. Enlaces duros: es una forma de darle un nombre alternativo a un archivo.
6. Enlaces simbólicos: son archivos de datos que contienen el nombre de otro archivo, para servir de enlace.

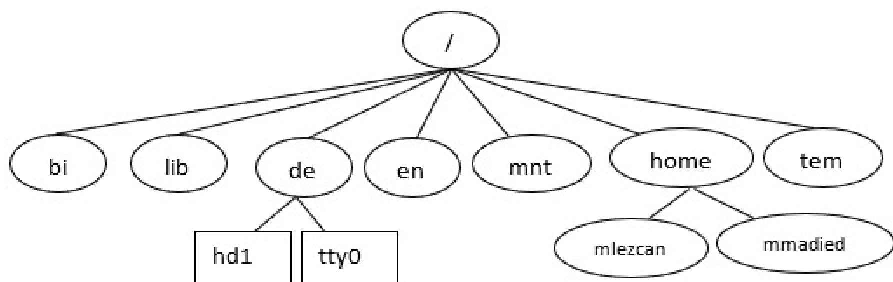


Figura 103. Estructura jerárquica de los directorios UNIX.

Algunos nombres de directorios de los SO tipo UNIX son comunes a todas las implementaciones; por ejemplo (figura 103):

- /bin. Contiene los comandos más comunes.
- /lib. Contiene bibliotecas de algunos lenguajes de programación (C en particular).
- /dev. Contiene manipuladores de dispositivos (device drivers) especiales que controlan el acceso a los periféricos. En la figura 95 se muestran los manipuladores: hd1, para manipular discos duros y tty0, para manipular terminales (Lezcano Brito, 2018).
- /entre otros. Contiene programas y archivos de datos del sistema. Los archivos en el directorio /entre otros/default contienen información que el sistema usa por defecto.
- /mnt. Es un directorio vacío reservado para montar sistemas de archivos.

- /home. Aloja los directorios de todos los usuarios del sistema y otros directorios que contienen comandos y archivos de datos adicionales. En la figura 95 los usuarios con cuentas en ese sistema son: mlezcano y mmadiedo.
- /tmp. Contiene archivos temporales creados por programas del SO. Los archivos están presentes cuando se está ejecutando el programa.

Control de acceso

Las primeras versiones de UNIX introdujeron una manera de controlar el acceso a los archivos que aún se mantiene y que se apoya en los aspectos siguientes:

- A cada usuario se le asigna un identificador único, conocido como UID (User identification).
- Cada usuario pertenece a un grupo primario, identificado con un GID (Group identification). Los usuarios pueden pertenecer a otros grupos adicionales.
- Cada vez que se crea un archivo se le asigna:
  - Un propietario (identificado por un uid) que típicamente es su creador,
  - Un grupo (identificado por un gid), puede ser el mismo de su creador o el del directorio padre si tiene fijado el permiso SetGID.

Los uid y gid de cada archivo residen en su nodo i.

- A cada archivo se le asocian 12 bits de protección.

Nueve de los 12 bits de protección especifican permisos de: lectura (read), escritura (write) y ejecución (execute) para tres categorías de usuarios:

- Su propietario (owner).
- El grupo al que pertenece (group).
- Los demás (others).

Se usará el comando ls para ejemplificar la idea. Este comando imprime en pantalla el listado del directorio actual o de una especificado como parámetro y admite diversas opciones (antecedidas por el signo -) que permiten obtener información diversa del directorio y sus archivos, una de ella es la opción l (long) que muestra información detallada de cada archivo.

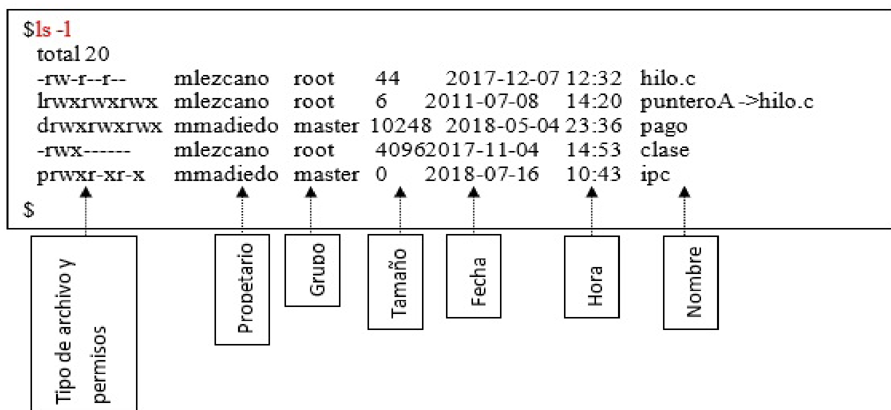


Figura 104. Listado del directorio actual.

En la figura 104 se muestra el resultado de ejecutar el comando ls -l, se ha resaltado esa orden en rojo para distinguirla de la salida del comando. El primer carácter de cada línea producida por ls identifica el tipo de archivo. A continuación, se detallan esos caracteres y se ponen ejemplos de acuerdo a la figura:

- El carácter - identifica los archivos ordinarios, ejemplos: hilo.c y clase (los nombres de los archivos aparecen al final de cada línea listada).
- La letra l se usa para los enlaces simbólicos, ejemplo: punteroA. Obsérvese el nombre del archivo en la forma: punteroA->hilo.c que especifica que punteroA es un enlace simbólico hacia el archivo físico hilo.c.
- La letra d simboliza los directorios, ejemplo: pago.
- La letra p distingue las tuberías nombradas, ejemplo: ipc.

Después de ese carácter aparecen los 9 bits de permiso mencionados anteriormente que se agrupan en tres ternas en la forma rwx para especificar los permisos sobre cada archivo:

De lectura (r), de escritura (w) y de ejecución (x); el carácter - en cualquier posición significa que no se tiene ese permiso.

Las ternas se usan de la manera siguiente:

- La primera define los permisos que tiene el propietario (owner) sobre el archivo.

- La segunda especifica los permisos que tienen los miembros del grupo (group) sobre el archivo.
- La tercera y última define los permisos de los restantes usuarios (others).

Se retoma la figura 96 para ejemplificar. En este caso, sobre el archivo hilo.c, que pertenece al usuario mlezcano del grupo root, se tienen los permisos: rw-r--r--, o sea el propietario puede leer y escribir, los miembros del grupo root y los demás usuarios solo pueden leer el archivo.

En realidad, los bits son 0 o 1: un 0 en una posición dada niega el permiso, un 1 lo otorga; o sea las rwx mencionadas son solo la vista externa de esos bits.

Los permisos aplicados a un directorio tienen el significado siguiente:

- El permiso de lectura (r) significa que se puede listar el directorio.
- El permiso de escritura (w) garantiza que se puedan crear, renombrar y borrar los archivos de ese directorio.
- El permiso de ejecución (x) permite hacer búsquedas dentro del directorio.

Los restantes tres bits (de los 12) definen otros detalles acerca de los archivos y directorios:

- El primer bit se denomina setuid o uid efectivo (effective user ID). Si un usuario ejecuta un archivo que tiene fijado este bit, se le otorgarán los derechos del propietario del archivo temporalmente.
- El segundo bit se nombra setgid o gid efectivo (effective group ID). Si un usuario ejecuta un archivo que tiene fijado este bit, temporalmente se le otorgan los derechos del grupo al que pertenece el propietario.

Los cambios de los bits setuid y setgid solo permanecen el tiempo que dure la ejecución del programa, habilitando la creación de privilegios que de otra manera serían inaccesibles.

El permiso setgid sobre un directorio hace que los archivos que se creen hereden el grupo del directorio. El bit setuid se ignora.

- El último bit se nombra sticky bit. Solo tiene significado para los directorios, si está fijado solo los propietarios de los archivos, dentro de ese directorio, pueden renombrarlos, moverlos o borrarlos. Resulta útil para el trabajo sobre directorios compartidos de manera temporal.

En los SO tipo Unix existe un usuario especial, conocido como el súper usuario, que está exento de las restricciones de acceso que se han mencionado en este acápite, todo programa que tenga fijado el bit setuid para el súper usuario deberá tener acceso irrestricto a los recursos del sistema operativo.

En los apartados siguientes se ofrece un pequeño resumen acerca de algunos sistemas de archivos soportados por los SO tipo UNIX, aunque no es objetivo del libro hacer un análisis amplio de ellos.

4.8.2. El sistema de archivo UFS

El sistema de archivo Unix File System (UFS) es un sistema de archivo derivado de FFS. Un volumen UFS está compuesto por (Lezcano Brito, 2018):

- Una cierta cantidad de bloques al comienzo de la partición que se reservan para iniciar el SO.
- Un súper bloque que contiene un "número mágico" para identificar el volumen como UFS y otros números que describen la geometría del volumen y otras características.
- Una colección de grupos de cilindros. Cada grupo de cilindros tiene los componentes siguientes (Lezcano Brito, 2018):
  - Una copia de resguardo del súper bloque.
  - Un encabezado con estadísticas del cilindro, incluyendo la lista de bloques libres del bloque, entre otros elementos.
  - Una cantidad de bloques para nodos i.
  - Una cantidad de bloques para datos.

Los nodos i se numeran secuencialmente. Los primeros están reservados por razones históricas (Lezcano Brito, 2018), después de ellos se localizan los nodos i del directorio raíz. Los metadatos se mantienen dentro de los nodos i.

Este sistema de archivo puede manipular discos de hasta 8 ZB (273 bytes), los nombres de archivo admiten 255 caracteres.

La mayoría de los proveedores de sistemas POSIX adaptaron UFS a necesidades propias y existen variantes de UFS implementadas por distintos propietarios, por ejemplo: SunOS/Solaris, System V Release 4, HP-UX,

Tru64 UNIX, 4.4BSD, BSD Unix systems FreeBSD (Lezcano Brito, 2018).

4.8.3. Sistemas de archivos extendidos (ext)

Los sistemas de archivos con el prefijo ext (extended file system) se crearon para los SO Linux y han tenido cuatro versiones: ext, ext2, et3 y ext4, que se han ido mejorando sucesivamente en el tiempo.

Sistema de archivo ext

El sistema de archivo ext (pudiera decirse primera versión del sistema de archivo ext, pero no es lo usual) fue el primer sistema que se creó específicamente para los SO Linux y se inspiró en UFS. Después de un tiempo fue reemplazado por otros dos sistemas de archivos: ext2 y xiaf (Lezcano Brito, 2018).

Sistema de archivo ext2

El ext2 (second extended file system) es un sistema de archivos para el *kernel* de Linux se usó en las distribuciones de Linux: Red Hat, Fedora Core y Debian. Se puede especificar el tamaño de los bloques cuando se crea el sistema de archivos, en un rango que fluctúa entre los 512 bytes y los 4 KiB, lo que resulta apropiado para ajustar ese tamaño a las necesidades del volumen y aprovechar mejor el espacio.

Sistema de archivo ext3

Una mejora significativa de ext3 (third extended file system) con relación a ext2 es que soporta el registro de archivos por diario (journaling), que no es más que una bitácora de las operaciones que se realizan sobre los volúmenes con el propósito de volver al pasado en caso de fallos, con ese propósito se realizan las acciones siguientes para cada transacción:

- Guarda el estado del volumen antes de la operación, de modo que si ocurre un fallo (de energía, del SO, entre otros.), pueda regresar al pasado (Lezcano Brito, 2018).
- Bloquea las estructuras de datos afectadas para que solo el proceso que hace la operación modifique dichas estructuras.
- Realiza las modificaciones anotando la manera de revertirla.
- Si falla la transacción, se deshacen los cambios, uno a uno, y se borra la traza guardada en el diario.
- Si se logra realizar la transacción, se borra el diario y se desbloquean las estructuras de datos involucradas.

El registro por diario tiene tres niveles:

- Diario (riesgo bajo): los metadatos y los archivos de contenido se copian al diario antes de llevarlos al sistema de archivo principal.
- Pedido (riesgo medio): solo se registran los metadatos en el diario, pero se asegura escribir el contenido del archivo en el disco antes que el metadato asociado se marque en el diario.
- Reescritura (riesgo alto): solo se registran los metadatos y el contenido del archivo se puede escribir antes o después que el diario se actualice.

Los sistemas ext3 utilizan un árbol binario balanceado (AVL) para agilizar las búsquedas e incorporan un mecanismo para asignar bloques de discos denominado Orlov, originario de BSD, para mejorar el rendimiento.

El sistema se puede actualizar a partir de ext2 sin perder datos y sin formatear, lo cual es una ventaja apreciable, pero a la vez es una desventaja porque la mayoría de sus estructuras de datos son iguales a ext2, lo que limita el uso de herramientas de diseño más actuales. El sistema no posee herramientas para desfragmentar. Este sistema llegó a ser el más usado en las distribuciones Linux, pero ya está siendo sustituido por ext4.

Sistema de archivo ext4

El sistema ext4 (fourth extended file system) es una mejora de ext3, que incluye varias capacidades nuevas, entre ellas los extends que reemplazan a los bloques usados en ext2 y ext3.

Un extend es un conjunto de bloques físicos contiguos que se pueden asignar como un todo para trabajar con archivos de gran tamaño, lo que reduce la fragmentación (Lezcano Brito, 2018).

Cualquier sistema ext3 puede montarse como ext4 sin necesidad de formatear el disco, dando una compatibilidad hacia atrás importante.

El sistema ext4, utiliza la técnica nombrada reserva retrasada que permite demorar la reservación de bloques hasta que la información esté a punto de escribirse, lo que mejora el rendimiento y reduce la fragmentación. El sistema también dispone de herramientas para desfragmentar archivos individuales o el sistema de archivo completo.

En esta sección se presentan tres ejemplos de programas que trabajan con algunas de las facilidades ofrecidas por el sistema de archivo. Los dos primeros programas están escritos en los lenguajes de programación, C y Bash. El objetivo es que el lector analice algunas particularidades de estos lenguajes. La pareja de programas tiene los mismos

objetivos, pero no se han programado exactamente iguales.

El tercer programa solo está escrito en C debido a que usan llamadas al sistema que deben invocarse desde ese lenguaje de programación.

La única tarea del primer programa, es imprimir el camino absoluto hacia el directorio de trabajo o actual. El programa C usa la función de biblioteca get_current_dir_name() pero también puede usarse la función gentre otroswd() en cualquiera de sus dos variantes (con uno o dos parámetros). El programa Bash, usa el comando externo pwd. (figura 105).

```c
//Ejemplo en C
#include <stdio.h>

void main()
  {
    char *workDir;

    workDir = (char *)get_current_dir_name();
    printf("El directorio de trabajo es: %s\n", workDir);
  }
```

```bash
#!/bin/bash
#Ejemplo en Bash
echo -n "El directorio de trabajo es: "
pwd
```

Figura 105. UNIX. Programa para imprimir el directorio actual o de trabajo.

```c
//Ejemplo en C
#include <stdio.h>
#include <stdlib.h>
#include <dirent.h>

void main(int argc, char *argv[])
  {
    DIR *directory;
    struct dirent* dir;

    if (argc != 2)
      {
        printf("Uso: %s <Nombre de directorio> \n", argv[0]);
      exit(1);
      }
    if ((directory = opendir(argv[1])) == NULL)
      {
        printf("No se pudo abrir el directorio: %s \n", argv[1]);
        exit(1);
      }
    printf("El directorio %s contiene los archivos siguientes:\n", argv [1]);
    while((dir = readdir(directory)) != NULL)
      printf("%s, con nodo i: %d\n", dir->d_name, (int)dir->d_ino);
    closedir(directory);
    exit(0);
  }
```

```bash
#!/bin/bash
#Ejemplo en Bash

if [ $# -ne 1 ]
  then
    echo "Uso: $0 <Directorio>"
   exit 1
  elif   ! [ -d $1 ]
    then
      echo "$1 no es un directorio"
      exit 1
fi
echo " Contenido del directorio $1"
ls -i $1
exit 0
```

Figura 106. UNIX. Programa para listar un directorio.

El segundo programa lista los archivos de un directorio que se pasa por parámetro (figura 106).

La versión en C utiliza la función opendir() para abrir un directorio que debe pasarse por parámetro (argumento argv[1]), la función readdir() obtiene un puntero a una estructura de tipo dirent que contiene la próxima entrada del directorio que se lea y la función closedir() cierra el directorio que se abrió.

La versión Bash verifica que el nombre pasado como argumento ($1) sea un directorio y si todo está bien, utiliza el comando externo ls con la opción -i (nodo i) para listar el contenido del archivo.

El programa de la figura 107 es el último de esta sección, utiliza las llamadas al sistema: open(), read() y write(). Debido a que estás llamadas están invocando funciones que están en el núcleo del SO resulta un poco más complejo el trabajo con ellas.

```
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#define SIZE 1024
main(int argc, char * argv[])
    {
        int readFile, writeFile;
        int readCount, writeCount;
        char buf[SIZE];
        char *bufPointer;
        if(argc !=3)
            {   printf("Uso %s readFile writeFile\n", argv[0]); exit(1);                          }
        if((readFile = open(argv[1], O_RDONLY)) == -1)
            { printf("No se pudo abrir %s", argv[1]); exit(1);                                    }
        if((writeFile = open(argv[2], O_WRONLY | O_CREAT )) == -1)
            { printf("No se pudo crear %s\n", argv[2]); exit(1);                                  }
        while(readCount = read(readFile, buf, SIZE))                    //Intenta leer SIZE bytes
            {
                if(readCount == -1) break;                              //Error. Saltar a próxima iteración
                else   if(readCount > 0)                                //Se leyó algo, pudiera ser menor que lo solicitado
                    {
                        bufPointer = buf;                               //El puntero para escritura apunta a lo leído
                        //Ciclo de escritura que intenta escribir readCount bytes en buf
                        while(writeCount = write(writeFile, bufPointer, readCount))
                            {
                                if(writeCount == -1) break;             //Error. Saltar a próxima iteración
                                else
                                    if (writeCount == readCount) break; //Se escribió todo lo leído
                                    else
                                        if (writeCount > 0)         //No se escribió todo
                                            {
                                                bufPointer += writeCount;   //Situar el puntero en el primer byte no escrito
                                                readCount -= writeCount;    //Esta cantidad de bytes no se han escrito
                                            }
                            }                                           //Hast que se escriba todo el búfer

                        if(writeCount == -1) break;   //Hubo algún error al escribir. Saltar
                    }
            }       //Hasta que llegue el fin de archivo de entrada
        /*Cerrar los archivos que se abrieron*/
        close(readFile);
        close(writeFile);
        exit(0);
    }
```

Figura 107. UNIX. Programa que usa llamadas al sistema.

El programa de la figura 107 pudiera hacerse de una forma más cómoda si se usan funciones de biblioteca de C, pero este es un libro de SO y las funciones de C que se utilicen, en la práctica, invocarán a estas mismas llamadas al sistema, aunque ese detalle quedará oculto al programador. A continuación, se detallan las partes fundamentales del programa:

- La llamada al sistema en la forma open(argv[1], O_RDONLY) abre, en lectura solamente (O_RDONLY), el archivo pasado como primer argumento (argv[1]). El valor devuelto es un descriptor de archivo que se asigna a la variable readFile.
- La llamada al sistema en la forma open(argv[2], O_WRONLY | O_CREAT) abre el archivo pasado como segundo argumento (argv[2]) en escritura solamente (O_WRONLY), si no existe lo crea (O_CREAT). El valor devuelto es un descriptor de archivo que se asigna a la variable writeFile.
- La llamada al sistema read(readFile, buf, SIZE) intenta leer SIZE bytes desde el archivo con descriptor readFile y los deposita en buf, devolviendo la cantidad real de datos leídos (se le asigna a readCount).
- La llamada al sistema write(writeFile, bufPointer, readCount) escribe en el archivo con descriptor writeFile una cantidad de datos readCount que están en la dirección apuntada por bufPointer, devolviendo la cantidad real de datos escritos que puede ser menor a readCount debido a: no había espacio en el medio, la llamada se interrumpe al recibir una señal, entre otros.

No se ofrecerán mayores detalles de estas facilidades. El lector puede consultar la ayuda, desde una terminal de su sistema UNIX. El comando man se usa con ese propósito, por ejemplo:

- Para conocer acerca de las funciones gentre otroswd (): man gentre otroswd.
- Para detalles acerca del comando pwd: man pwd.
- Para conocer acerca de la llamada al sistema write() será necesario especificar la sección del manual en que está (la 2), debido a que existe un comando con igual nombre que está en la sección 1 del manual y por defecto se ofrece la ayuda del primero que se encuentre, de manera que deberá usarse la sintaxis siguiente: man -s 2 write.

4.9. Los SO Windows y sus sistemas de archivo

Los SO Windows utilizan fundamentalmente dos sistemas de archivos: File

Allocation Table (FAT) y New Technology File System (NTFS). El sistema FAT data de la época del SO MS-DOS y aún se usa, sobre todo en soportes destinados a realizar intercambios con otros sistemas, como las memorias flash.

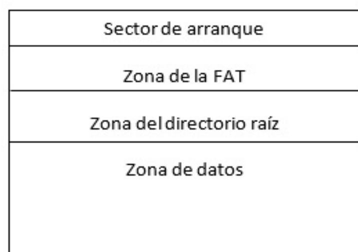


Figura 108 Estructura de un disco con formato FAT.

Un disco formateado con este sistema tiene la estructura que se muestra en la figura 108:

- El sector de arranque (primero de la partición o volumen) incluye información básica del sistema de archivo y el código de carga del SO.
- La zona de la FAT está dedicada a dos copias de una estructura de datos especial, denominada FAT (se explica más adelante).
- La zona del directorio raíz está ocupada por la tabla del directorio raíz.
- La zona de datos se dedica al contenido de los archivos.

Nombre (8 bytes)	Extensión (3 bytes)	Atributos (1 byte)	Reservado (10 bytes)	Hora (2 bytes)	Fecha (2 bytes)	Clúster inicial (2 bytes)	Longitud (4 bytes)
notas doc						21	
pipe c						11	
semaphore	c					50 7	

Figura 109. Tabla de directorio (original) del sistema FAT.

Este sistema no ofrece mecanismos reales para la seguridad y utiliza las estructuras de datos tabla de directorio y FAT para localizar los archivos.

La estructura de la tabla de directorio (figura 109) ha cambiado con el tiempo, pero, en esencia, mantiene un formato que permite almacenar información acerca de los archivos, tales como:

- Nombre y extensión del archivo: estos dos campos combinados identifican al archivo, es usual referirse a ella como 8.3, significando que los nombres de archivos tienen dos partes: la primera para el nombre (8 caracteres) y la segunda para la extensión (3 caracteres). A partir de Windows 95 el nombre de los archivos se extendió a 255 caracteres.

La idea de la extensión se mantiene en NTFS.

- Atributos: especifica algunas propiedades de los archivos, tales como: solo lectura, oculto, de sistema, volumen, directorio y archivo.
- Hora y fecha: permiten saber cuándo se modifican los archivos, es útil para las actualizaciones.
- Longitud: tamaño del archivo en bytes.
- El campo identificado como clúster inicial indica cuál es el primer clúster de cada archivo y se usa como una entrada en la FAT para localizar los bloques restantes que pertenecen al archivo.

	0	1	2	3	4	5	6	7	8	9
0	Reservados		23	4	6	8	eof	0	2	0
1	0	3	0	0	0	0	0	0	0	0
2	0	5	0	34	0	0	0	0	0	0
3	0	0	0	0	40	0	0	0	50	0
4	eof	0	0	0	0	0	0	0	0	0
5	eof	0	0	0	0	bad	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0

Figura 110. El FAT.

La figura 110 presenta la tabla FAT en forma de una matriz cuadrada, con espacio para referenciar solo 99 clúster (en la práctica es posible referenciar más). Debe observarse la numeración asociada a los clústeres para facilitar la explicación: la primera fila refiere a los clústeres que van desde el 0 hasta 9, la segunda desde el 10 hasta el 19 y así sucesivamente hasta la última que abarca las referencias desde el clúster 90 hasta el 99.

La tabla FAT tiene dos objetivos:

- Conocer la cadena de bloques que conforman un archivo.
- Controlar los clúster libres y ocupados.

Seguidamente, y a manera de ejemplo, se analizan las figuras 109 y 110 de forma conjunta. A partir de la figura 110, puede determinarse que hay tres archivos en la unidad referida: notas.doc que comienza en el clúster 21, pipe.c que inicia en el clúster 11 y semaphore.c que comienza en el clúster 50.

Los clústers iniciales de cada archivo, tomados a partir del campo correspondiente de la tabla de directorio de la figura 110, se toman como referencias o apuntadores a la tabla FAT (figura 110), cada posición de la FAT indica cuál es el siguiente clúster del archivo; si es el último se señala con una marca especial.

Con estas ideas en mente es fácil encontrar la cadena de clúster que conforman cada archivo, en el ejemplo:

- notas.doc, formado por los clústers: 21, 5, 8, 2, 23, 34 y 40.
- pipe.c, constituido por los clúster:s 11, 3, 4 y 6.
- semaphore.c, que solo tiene un bloque, el 50.

La última referencia a bloques del archivo se marca con un número especial, representado por eof (end of file, es la marca que por convenio se pone en los textos, pero en realidad es un número especial) en la figura. Los bloques finales son:

- 40 para el archivo notas.doc.
- 6 para el archivo pipe.c.
- 50 para el archivo semaphore.c.

Los espacios de la FAT que contienen el número 0 se usan para identificar los clústers que están libres; por ejemplo, los clústers 7, 9, 10, 12,

entre otros. de la figura 28 están libres. El clúster 55 está dañado y se ha indicado con la palabra inglesa bad, aunque en realidad también es un número especial.

La tabla FAT es un gran bloque de índices, para todos los archivos del volumen, y a la vez es una tabla de clúster libres. Resulta catastrófica la pérdida o el daño de la FAT, debido a que ella es la única estructura de datos que controla todo el volumen de almacenamiento y aunque el sistema conserva una copia de la FAT para evitar los posibles desastres, esta solución nunca arrojó muy buenos resultados.

El tamaño de los bloques de la tabla FAT determina la cantidad de clúster de disco que puede referenciarse. Las versiones de este sistema se han distinguido por ir acompañada de un número n, que especifica ese tamaño, han existido las siguientes implementaciones:

FAT12

Este sistema de archivo data del año 1982 y permite referenciar 212 (4096) clúster, aunque se concibió como un sistema de archivo para disquetes fue suficiente para dar soporte a los discos de 10 MB que surgieron posterior a su lanzamiento usando clúster de 8 kilobytes, en realidad el tamaño permitido era de 32 MB.

FAT16

El sistema estuvo disponible en 1988, para la versión 4.0 de MS-DOS, permitía referenciar 216 clúster y manejar discos duros de hasta 90 GiB.

Windows XP aumentó el tamaño de los espacios de la FAT a 64 (FAT64) tomando la cuenta del clúster como un entero sin signo, pero este formato no era compatible con otras implementaciones de aquellos años y también provocaba más fragmentación interna.

Windows 98 fue compatible con la versión FAT64, pero las utilidades de disco no podían trabajar con esta versión.

FAT32

La FAT32 superó los límites de FAT16 y mantuvo la compatibilidad con MS-DOS en modo real, esta versión permite manejar discos de hasta 4 GiB.

De las tres versiones de FAT mencionadas, esta es la única que se mantiene hoy en día. Han existido otras versiones de la FAT, denominadas VFAT (Windows 3.11) y FASTFAT (Windows NT 3.1). También existe extFAT como una extensión de la FAT.

Algunos SO Windows no soportan implementaciones de la FAT, convirtiendo este sistema de archivo en un medio adecuado para el intercambio de datos entre distintos SO.

4.9.1. El sistema de archivo NTFS

NTFS es un sistema de archivo elegante y flexible que se destaca por los aspectos que a continuación se relacionan:

- Es capaz de recuperarse de fallas del sistema y de los discos, para lo cual reconstruye el volumen retornando a un estado consistente.

Para poder hacer la reconstrucción, el sistema realiza todos los cambios en forma atómica (se hacen totalmente o no). De esta manera si una acción falla, puede revertirse (regresar atrás en el tiempo).

Por otra parte, el sistema almacena los datos críticos en forma redundante, así no hay peligro de pérdidas.

- Con relación a la seguridad se usa un modelo en el cual un archivo abierto se implementa como un objeto archivo con un descriptor de seguridad que define sus atributos y persiste, como un atributo, para cada archivo sobre el equipo de almacenamiento.
- Pueden manipularse archivos y discos muy grandes y de manera eficiente.
- El contenido real de un archivo se trata como un torrente (stream) de bytes y es posible definir múltiples torrentes de datos para un archivo dado.
- Permite registro de diario o bitácora (journaling).
- Los directorios y los archivos se pueden comprimir y encriptar de manera transparente.
- Soporta enlaces duros o sea permite que un archivo pueda accederse desde distintos caminos o rutas (path) dentro del mismo volumen, también soporta enlaces simbólicos para permitir el acceso a archivos y directorios desde múltiples caminos aun cuando estén en diferentes volúmenes. Lo que lo hace compatible con POSIX.
- Brinda la facilidad de los puntos de montajes (típico de UNIX) lo que permite que un volumen dado aparezca unido a la estructura de

directorio de otro sin necesidad de incluir la letra que lo identifica, esto último característico de Windows.

El clúster, uno o varios sectores, es la unidad mínima de asignación en un volumen NTFS y su tamaño se establece en el momento en que se formatea lógicamente el volumen, esto quiere decir que las asignaciones de espacio a los archivos se hacen a nivel de clúster, los cuales no necesitan estar contiguos.

Organización de un volumen NTFS

En NTFS todos los elementos de un volumen son archivos y cada uno de ellos es una colección de atributos, a tal extremo que hasta los datos se tratan como atributos. Esta estructura tan simple permite establecer funciones generales para organizar y manejar los volúmenes.

Figura 111. Estructura de volúmenes NTFS.

Un volumen NTFS está formado por cuatro regiones (figura 111):

- Primero se sitúa el sector de carga de la partición (en realidad no es un sector y puede medir hasta 16 sectores), que contiene información acerca del volumen y la estructura del sistema de archivo. También contiene información y código para arrancar el SO.
- El segundo lugar lo ocupa la tabla de archivo maestro, más conocida por MFT (Master File Table), ella contiene información acerca de todos los archivos y directorios del volumen. En esencia es una tabla que contiene archivos y atributos organizados en fila.
- El tercer puesto está destinado a los archivos del sistema, entre ellos deben mencionarse los siguientes:
  - MFT2, una copia de las primeras filas de MFT usada como garantía de seguridad ante fallos de esta tabla.
  - El archivo log, Una lista de transiciones que se usa para recuperar el volumen NTFS.
  - Mapa de bit de los clústers, que muestra los clústers que están en uso.

- Tabla de definición de atributos, define los tipos de atributos soportados por el volumen, indicando si pueden indexarse y recuperarse durante una operación de recuperación.

- El último lugar está destinado a los archivos en sí.

MFT

La MFT es una tabla de registros organizados por filas y es el centro del sistema de archivo NTFS, cada fila describe un archivo del volumen, lo que incluye al propio MFT.

Cuando los archivos son pequeños todo su contenido se coloca en el mismo registro MFT que le corresponda, si no es así el registro solo contiene información parcial del archivo y desde ahí se apunta a otros clústers del volumen que contienen el resto de la información del archivo.

Cada registro en la tabla MFT está formado por un conjunto de atributos que se usan para definir las características del archivo o directorio, estos son:

- Información general: atributos de acceso, marca de tiempo, cantidad de apuntadores al archivo.
- Lista de atributos: la lista de atributos que conforman el archivo y su localización. Se usa cuando un archivo no cabe dentro del registro MFT.
- Nombre del archivo o directorio (puede tener más de uno).
- Descriptor de seguridad: especifica el propietario del archivo y quiénes pueden acceder a él.
- Datos: contenido del archivo.
- Índice raíz: se utiliza para implementar directorios.
- Información del volumen: puede ser, entre otras, su nombre y versión.
- Mapa de bit. Mapa de los registros usados sobre MFT o directorio.

Resumen del capítulo

Conceptualmente los volúmenes son discos lógicos y el sistema de archivo es la parte del SO que tiene la misión de administrarlo, presentando a los usuarios el concepto abstracto de archivo sin obligarlos a conocer la forma en que se guarda la información sobre dichos equipos.

Los archivos pueden agruparse en directorios para poder localizarlos más fácilmente y proporcionar una visión más organizada de cada vo-

lumen.

El sistema de archivo puede verse como un conjunto de capas jerárquicas que van desde el nivel inferior, encargado de controlar la entrada/salida básica o a nivel de los equipos; hasta el nivel de entrada/salida lógica donde tiene sentido el concepto de archivo.

El sistema de archivo también debe mantener el control de los espacios libres y ocupados de cada volumen. Se usan tres técnicas para la asignación de espacio: contigua, enlazada e indexada.

Para asignar espacios puede usarse una política de pre asignación, que exige conocer la longitud que tendrá cada archivo o puede usarse asignación dinámica de acuerdo a cada demanda.

Para controlar el espacio libre se pueden usar varias técnicas, entre las que deben mencionarse las siguientes: los mapas de bits que usan un vector para controlar cada bloque de un volumen, las tablas de espacios libres que refieren a los espacios vacíos por medio de una dirección de inicio y una longitud, las listas de bloques libres que asignan un número secuencial a cada bloque manteniendo la lista de todos los bloques libres en una parte reservada del equipo de almacenamiento.

Existen diversos sistemas de archivos, algunos que deben destacarse son los sistemas: FAT y NTFS de los SO Windows, Unix File System (UFS) y sus múltiples derivados en los SO tipo Unix, entre ellos los ext que son específicos de los SO Linux.

4.10. Ejercicios propuestos

1. Explique el concepto de bloque y justifique el hecho de que el espacio en un volumen no se asigne byte a byte.

2. Tomando en cuenta la forma que adoptan el directorio y la FAT del sistema de archivo homónimo. Diseñe algoritmos para:

a) Borrar archivos.

b) Mover archivos.

c) Copiar archivos.

3. Defina las estructuras de datos imprescindibles para implementar un sistema de archivo de colocación contigua. Justifique su respuesta.

a) Diseñe un algoritmo para borrar archivos en ese sistema.

4. **Defina las estructuras de datos imprescindibles para implementar un sistema de archivo de colocación enlazada. Justifique su respuesta.**

a) Diseñe un algoritmo para mover archivos en ese sistema.

5. **Defina las estructuras de datos imprescindibles para implementar un sistema de archivo de colocación indexada. Justifique su respuesta.**

a) Diseñe un algoritmo para copiar archivos en ese sistema.

6. **Explique la importancia de la estructura de datos MFT en el sistema de archivo NTFS.**

7. **Explique la importancia de los nodos i en los sistemas de archivo tipo UNIX.**

8. **Los SO vienen acompañados por un conjunto de utilitarios conocidos como comandos, que pueden clasificarse en internos y externos.**

a) Explique las diferencias entre los comandos internos y los externos.

b) Busque algunos comandos internos y externos de los SO Windows y explique sus propósitos. Algunos de ellos los puede encontrar en el anexo 2.

c) Busque algunos comandos internos y externos de los SO tipo UNIX y explique sus propósitos. Algunos de ellos los puede encontrar en el anexo 1.

9. **Amplíe sus conocimientos acerca de los sistemas de archivos ext, haga una tabla de diferencias entre: ext, ext2, ext3 y ext4.**

10. **¿Por qué se afirma que la pérdida del FAT en los sistemas de archivos del mismo nombre es un hecho catastrófico para el volumen?**

a) ¿Cree usted que resulte adecuada la estrategia de mantener este tipo de estructuras especiales en lugares específicos de un volumen? Explique.

11. **Los sistemas de archivos de colocación contigua necesitan de algún mecanismo para desfragmentar los volúmenes. Explique los pasos generales que debe seguir un algoritmo para realizar esta tarea.**

a) ¿Cree usted que sea imprescindible ese mecanismo en los sistemas de colocación enlazada e indexada? Justifique su respuesta.

b) ¿Cree usted que sea recomendable ese mecanismo en los sistemas de colocación enlazada e indexada? Justifique su respuesta.

Referencias bibliográficas

Ap Morbach, R., & Annoni Pazeto, T. (2017). Proposta de um simulador para o ensino de escalonamento FIFO e DRR. (Ponencia). III Congreso Internacional de Computación y Telecomunicaciones. Campinas, Brasil.

Bernstein, A. J. (1966). Analysis of programs for parallel processing. IEEE Transactions on Electronic Computers, 15(5), 757-763.

Coffman, E. G., Elphick, M. J., & Shoshani, A. (1971). Sistemas Operativos. Computing Surveys, 3(2), 68-78.

Correia Barbosa, M. B. (2019). Garantizando el acceso concurrente de múltiples procesos a un buffer limitado. Universidad & Ciencia, 8(2), 253-267.

Dawwd, S. (2019). GLCM based parallel texture segmentation using a multicore processor. The International Arab Journal of Information Technology, 16(1).

García Chango, S. X., & Salazar Ramón, L. X. (2018). Uso de la memoria virtual en los sistema operativo modernos. Universidad de las Fuerzas Armadas.

Hernández, R., Alvarado, E., & Escarcega, L. (2017). Implementación del algoritmo el menos recientemente usado (LRU) en la. Revista de Tecnología Informática, 1(1), 52-60.

Hoare, C. A. (1974). Monitors: An Operating System Structuring Concept. Communications of the ACM, 17(10), 549–557.

Lezcano Brito, M. G. (2017). Administración de la memoria. En, M. G. Lezcano Brito, Fundamentos de sistemas operativos. Entornos de trabajo. (pp. 105-127). Ediciones Universidad Cooperativa de Colombia.

Lezcano Brito, M. G.(2018). Fundamentos de sistemas operativos. Entornos de trabajo. Ediciones Universidad Cooperativa de Colombia.

Millo Sánchez, R., Paz Rodríguez, W., Gallardo Segura, A., & López León, H. (2016). XEOS para el desarrollo de software base de sistemas. Revista Cubana de Ciencias Informáticas, 10(2).

Peterson, G. (1981). Myths about the mutual exclusion problem. Information Processing Letters, 12(3), 115-116.

Silberschatz, A. (1996). Operating System Concepts, 5th Edition.

Silberschatz, A., Galvin, B., & Peter, G. (2006). Fundamentos de sistemas operativos. Madrid: McGraw-HILL/Interamericana.

Sol Llaven, D. (2015). Sistemas Operativos: Panorama para ingeniería en computación e informática. Grupo Editorial Patria S.A de C.V.

Stallings, W. (2014). Operating systems: Internals and design principles. Pearson.

Vázquez Carmona, E. V., Vázquez López, R., & Herrera Lozada, J. C. (2019). Desarrollo de un simulador para algoritmos de sustitución y escritura de memoria caché. Pistas Educativas, 41(135), 45-59.

Anexos

Anexo 1. Comandos Unix.

El anexo 1 es un complemento del libro, que se ofrece como guía inicial para los lectores interesados en profundizar acerca de las facilidades que ofrecen los comandos de los SO tipo Unix. Solo se referencian algunos comandos del bash que, a juicio de los autores, son los más usados.

El Shell es un programa que permite la interacción entre los usuarios y el SO y se inicia, como un proceso normal, cada vez que un usuario entra al sistema.

El Shell que se usa depende de la configuración de cada usuario. El archivo /entre otros/passwd contiene una lista de todas las cuentas y algunos detalles de configuración. Cada entrada de ese archivo tiene siete campos separados por el símbolo de dos puntos (:), que especifican diferentes facilidades para los usuarios, ejemplo:

mlezcano:x:1002:1002:Mateo Lezcano,,,,:/home/mlezcano:/bin/bash

El último campo especifica el Shell que se usa (bash en este caso) y el camino en que se encuentra (/bin).

Los comandos se pueden teclear en forma interactiva o incluirse dentro de un programa script.

Comandos internos

Los comandos internos forman parte del código del Shell, en esta sección se muestran algunos de los que acompañan al Bash.

alias Define un sobrenombre a alias para un comando especifico.

bg Envía un trabajo (job) a ejecutarse como proceso de fondo.

cd Cambia el directorio actual hacia el directorio especificado.

exit Sale del Shell.

fg Envía un trabajo (job) a ejecutarse como proceso de superficie.

jobs Lista los trabajos (jobs) activos.

logout Sale del Shell.

read Lee una línea desde la entrada estándar y se le asigna a una

variable.

return Sale de una función.

umask Fija los permisos por defecto para los archivos y directorios que se creen.

unalias Elimina el alias especificado.

unset Elimina las variables de entorno o los atributos del Shell especificados.

wait Espera que el proceso especificado termine.

Comandos externos

Los comandos externos son programas que se instalan junto con el SO. Esta sección muestra algunos de ellos, caso todos se localizan en el directorio /bin.

cat Concatena los archivos especificados e imprime su contenido por la salida estándar.

chgrp Cambia el grupo por defecto del archivo o directorio especificado.

chmod Cambia los permisos sobre el archivo o directorio especificado.

chown Cambia el propietario del archivo o directorio especificado.

chpasswd Lee un archivo conteniendo pares en la forma login:password y actualiza las palabras claves (password) de los usuarios especificados en la parte login.

clear Limpia la pantalla de la terminal y sitúa el cursor en su extremo superior izquierdo.

cp Copia archivos y directorios.

cut Elimina secciones de cada línea del archivo especificado.

date Fija o muestra la fecha y la hora del sistema.

df Reporta el uso del espacio de los sistemas de archivos.

echo Muestra la cadena especificada por la terminal de salida es-

tándar.

find Realiza una búsqueda recursiva del archivo especificado.

free Ofrece un reporte de la memoria utilizada.

groupadd Crea un grupo nuevo.

kill Envía una señal del sistema al proceso especificado.

less Permite mostrar un archivo por pantallas que se detienen esperando una acción del usuario.

link Crea un enlace hacia un archivo usando un alias.

ls Lista el contenido de un directorio.

man Muestra un texto de ayuda acerca del tópico especificado.

mkdir Crea un directorio.

more Similar a less pero menos eficiente

mount Monta un equipo en el sistema de archivo.

mv Mueve o renombra un archivo.

nice Ejecuta un comando con una prioridad modificada.

ps Muestra información sobre los procesos.

pwd Muestra el camino absoluto hacia el directorio de trabajo actual.

rm Borra el archivo especificado.

rmdir Borra el directorio especificado.

sleep Pausa una operación por el tiempo especificado.

sort Ordena las líneas de un archivo texto.

stat Muestra estadísticas acerca de un archivo dado.

sudo Ejecuta una aplicación como el usuario root.

top Muestra estadística del sistema.

touch Crea un archivo vacío.

umount Desmonta un equipo del sistema de archivo.

uptime Muestra información acerca del tiempo que ha estado ejecutando el sistema.

useradd Crea una cuenta de usuario.

userdel Elimina una cuenta de usuario.

usermod Modifica una cuenta de usuario.

vmstat Emite un reporte acerca del uso de la memoria virtual y de la CPU, entre otros detalles.

which Encuentra la localización de un comando.

who Muestra los usuarios que están conectados al sistema.

whoami Muestra el usuario actual.

Anexo 2. Comandos Windows.

El anexo 2 es un complemento del libro, que se ofrece como guía inicial para los lectores interesados en profundizar acerca de las facilidades que ofrecen los comandos de los SO Windows. Solo se referencian algunos comandos que, a juicio de los autores, son los más usados.

Los comandos Windows pueden usarse de manera interactiva, desde el interprete de comandos o dentro de programas script, con extensión bat, para realizar diversas tareas.

Comandos internos

assoc	Muestra o modifica las asociaciones de extensiones de archivos.
cd	Cambia o muestra el directorio actual.
cls	Borra la pantalla.
copy	Copia uno o más archivos en otra ubicación.
date	Muestra o establece la fecha.
del	Elimina uno o más archivos.
dir	Lista el contenido de un directorio.
echo	Muestra mensajes, o activa y desactiva el eco.
md	Crea un directorio.
mklink	Crea vínculos simbólicos y físicos.
more	Muestra la información pantalla por pantalla.
move	Mueve uno o más archivos de un directorio a otro.
rd	Elimina un directorio.
ren	Cambia el nombre de uno o más archivos.
rmdir	Elimina un directorio.
type	Muestra el contenido de un archivo de texto.
vol	Muestra la etiqueta del volumen y el número de serie del disco.

Comandos externos

Los comandos externos que se refieren se localizan dentro del directorio

Windows\System32 de una partición de arranque del SO Windows 10, típicamente la unidad C.

attrib Muestra o cambia los atributos de un archivo.

chkdsk Comprueba un disco y muestra un informe de estado.

chkntfs Muestra o modifica la comprobación del disco en el tiempo de arranque.

clip Redirecciona el resultado de las herramientas de la línea de comandos

cmd Inicia una nueva instancia del intérprete de comandos.

comp Compara el contenido de dos archivos o conjuntos de archivos.

compact Muestra o altera la compresión de los archivos en particiones NTFS.

convert Convierte un volumen FAT a NTFS.

defrag Optimiza y desfragmenta archivos en volúmenes locales.

fc Compara dos archivos o conjuntos de archivos y muestra las diferencias.

find Busca una cadena de texto en uno o más archivos.

format Formatea un disco con el sistema de archivo especificado.

help Ofrece una ayuda resumida acerca de todos los comandos.

hostname Muestra el nombre del host actual.

nbtstat Muestra estadísticas del protocolo y las conexiones actuales TCP/IP

taskkill Termina o interrumpe un proceso o aplicación que se está ejecutando.

tasklist Muestra todas las tareas en ejecución, incluidos los servicios.

tree Muestra gráficamente la estructura de directorios de una unidad.

usando NBT (NetBIOS sobre TCP/IP).

ver Muestra la versión de Windows.

xcopy Copia archivos y árboles de directorios.

Anexo 3. Unidades de información.

El anexo 3 pretende arrojar un poco de luz sobre las confusiones que existen acerca de las unidades que se utilizan para medir cantidades de información en el mundo informático.

Tradicionalmente estas unidades de medidas se han basado en el sistema de numeración binario, en el cual solo existen los dígitos 0 y 1, lo que permite representar dos estados perfectamente distinguibles. El bit (binary digit) es un dígito en ese sistema y constituye la unidad mínima de información.

Una secuencia de bits puede codificar cualquier valor discreto tales como: números, palabras, imágenes, entre otros. y en general con un número de n bits se pueden representar hasta 2n valores o estados diferentes.

A partir de la cantidad de bits se han definido diferentes nombres para cada unidad de medida, hoy en día se acepta que un byte son 8 bits, pero en algunas computadoras antiguas podía estar conformado por 6, 7, 8 o 9 bits. En español se usa la palabra octeto para referirse a 8 bits exactos.

Para expresar unidades de medidas mayores se utilizaron, y se utilizan, términos inapropiados provenientes del sistema decimal; por ejemplo, las palabras kilo y mega para referirse a 1024 (210) y 1048576 bits (220) respectivamente. Desde ese momento comenzó la confusión porque en realidad kilo significa mil (103) y mega un millón (106).

Para tratar de enmendar este error histórico y arrojar un poco de claridad en esta problemática se publicó un apéndice al estándar IEC 60027-2 (en 1998) que introdujo el prefijo bi, delante de byte, para denotar las unidades que se corresponden con el sistema binario, dejando las palabras kilo, mega, entre otros. con sus significados reales en el sistema decimal. Infortunadamente esa nomenclatura no ha sido adoptada por toda la comunidad informática y el hecho de existir mucha bibliografía anterior al año 1998 también hace que haya bastante confusión en este aspecto.

Tabla A.III Unidades de información

Decimal			Binario		
Palabra	Símbolo	Cantidad	Palabra	Símbolo	Cantidad
Kilobyte	kB	103	Kibibyte	KiB	210
Megabyte	MB	106	Mebibyte	MiB	220
Gigabyte	GB	109	Gibibyte	GiB	230
Terabyte	TB	1012	Tebibyte	TiB	240
Petabyte	PB	1015	Pebibyte	PiB	250
Exabyte	EB	1018	Exbibyte	EiB	260
Zettabyte	ZB	1021	Zebibyte	ZiB	270
Yottabyte	YiB	1024	Yobibyte	YiB	280

Índice

Introducción 7
Capítulo I. Procesos e hilos 30
1.1. Los procesos en un equipo de cómputo 30
1.2. Paralelismo 33
1.3. Planificación de procesos 35
1.4. Control de procesos 47
1.5. Tipos de hilos 50
1.6. Concurrencia 54
1.7. Interbloqueo (deadlock) 73
1.8. Operaciones sobre procesos 83
1.8.1. Uso de llamadas al sistema en los SO tipo Unix...... 86
1.9. Ejercicios 95
Capítulo II. Administración de la memoria................................ 97
2.1. La memoria y su estructura 97
2.2. Particiones fijas 99
2.3. Particiones variables 102
2.4. Paginado 106
2.5. Segmentado 110
2.6. Memoria virtual 114
2.5.1. Memoria virtual paginada 117
2.5.2. Memoria virtual segmentada 121
3.5.3. Uso de información común. La protección..................... 124
2.7. Ejercicios propuestos 136

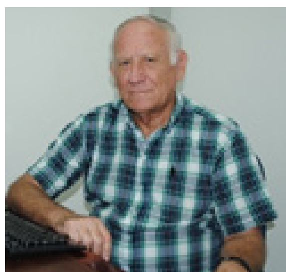
Capítulo III. Equipos de almacenamiento masivo 139
3.1. El sistema de archivos 139
3.2. Los discos magnéticos 139
3.1.1. Algoritmos de planificación de discos magnéticos.... 141
3.1.2. El formato de los discos 145
3.3. Conjunto redundante de discos independientes 149
3.4. Discos ópticos (CD y DVD) 150
3.5. Memoria USB 152
3.6. Unidades de estado sólido 153
3.7. Ejercicios propuestos 155
Capítulo IV. Sistema de archivos 156
4.1. El sistema de archivos y sus características 156
4.2. Los archivos 156
4.3. La arquitectura del sistema de archivos 157
4.4. Directorios 158
4.5. Compartir archivos 161
4.6. Métodos de asignación 162
4.6.1. Asignación contigua 163
4.6.2. Asignación enlazada 166
4.6.3. Asignación indexada 167
4.7. Manejo de los espacios libres 168
4.8. Sistemas de archivo de los SO tipo Unix 170
4.8.1. Antecedentes y generalidades 170
4.8.2. El sistema de archivo UFS 178
4.8.3. Sistemas de archivos extendidos (ext) 179

4.9. Los SO Windows y sus sistemas de archivo 183
4.9.1. El sistema de archivo NTFS 186
4.10. Ejercicios propuestos 191
Referencias bibliográficas 193
Anexos 195

Datos Biográficos

Miguel Angel Fernández Marín

Ingeniero en Ciencias Informática y Máster en Bioinformática y Bilogía Computacional. Fue profesor con categoría de Asistente en la Universidad de las Ciencias Informáticas de Cuba (UCI) y actualmente es profesor con categoría de Agregado 1 en la Universidad Metropolitana del Ecuador (UMET). Tiene 13 años de experiencia como profesor universitario, y en el transcurso ha colaborado con varias instituciones extranjeras. En la UCI fue Jefe de Departamento de Matemática, en la UMET colaboró como profesor, director de la carrera de Sistemas y actualmente es el Asesor del Vicerrector Académico. Tiene publicaciones en revistas científicas y memorias de eventos. Ha impartido diversos cursos en pregrado y postgrado.



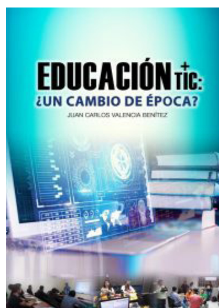
Mateo Gerónimo Lezcano Brito

Graduado de Cibernética-Matemática, con maestría en Informática Aplicada y doctorado en Ciencias Técnicas. Tiene 70 publicaciones en revistas científicas y memorias de eventos. Ha publicado siete libros y monografías. Profesor Titular y Profesor Consultante de la Universidad Central "Marta Abreu" de Las Villas de Cuba. Fue miembro de la dirección nacional de la Sociedad Cubana de Matemática y Computación (SCMC) por más de 15 años y es Miembro Emérito de esa Sociedad Científica, la cual le otorgó el premio Pablo Miquel por su sostenida labor en la Ciencia de la Computación y en la enseñanza de esa especialidad. Ha participado en 113 eventos científicos. Fue miembro permanente del Tribunal Nacional para el Otorgamiento del Título de Doctor en Matemática de la República de Cuba.

Zoila Zenaida García Valdivia

Licenciada en Matemática con doctorado en Ciencias Técnicas. Profesora Titular y Profesora Consultante de la Universidad Central "Marta Abreu" de Las Villas (UCLV) de Cuba. Tiene 47 años de trabajo en la UCLV y ha colaborado como profesor invitado en varias universidades, incluida la Universidad Metropolitana de Ecuador (UMET). Fue miembro del tribunal permanente para la defensa de doctorados en Ciencias Técnicas (Automática y Computación). Perteneció al Consejo Científico Ramal de Pedagogía de la UCLV. Formó parte de la dirección nacional de la Sociedad Cubana de Matemática y Computación, la cual le otorgó el premio Pablo Miquel por su labor en la enseñanza de la Ciencia de la Computación. Tiene publicaciones en revistas científicas y memorias de eventos, autora de dos libros. Ha impartido diversos cursos en pregrado y postgrado.

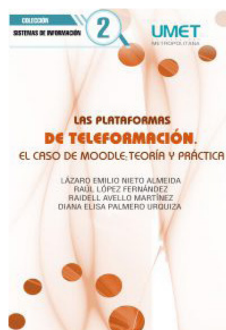
OTROS TÍTULOS DE NUESTRA EDITORIAL



Educación + TIC: ¿un cambio de época?

Juan Carlos Valencia Benítez

ISBN: 978-959-257-632-2



Las plataformas de teleformación. El caso de Moodle teoría y práctica.

Lázaro Emilio Almeida Nieto, Raúl López Fernández, Raidell Avello Martínez, Diana Elisa Palmero Urquiza

ISBN: 978-959-257-517-2

Los sistemas operativos, constituyen el programa fundamental del computador que permite relacionar la operatividad de la máquina con los usuarios a través de una interfaz de fácil aprendizaje y uso. Por ser ellos una parte esencial de cualquier sistema de cómputo, se estudian en las carreras de perfil informático como Ingeniería en Sistemas, Ingeniería en Ciencias Informáticas o Licenciatura en Ciencias de la Computación, siendo base para el alumno en sus conocimientos de la informática. Este libro pretende, explicar con detenimiento conceptos relacionados con las formas de planificar el uso del procesador central, los algoritmos que se usan para distribuir el tiempo de procesamiento y el tratamiento de los procesos concurrentes. De la misma forma, se analiza la administración de la memoria, el paginado y la segmentación vinculando los algoritmos y las estructuras de datos más utilizados en las tecnologías actuales. También se introducen los conceptos sobre los equipos de almacenamiento masivo, así como la estructura y organización de los sistemas de archivo en el computador y sus operaciones sobre los distintos soportes de almacenamiento. Además, cada capítulo propone un conjunto de ejercicios para ser resuelto por los lectores, los cuales sistematizan lo estudiado. Cuenta con anexos donde se explicitan los comandos más utilizados en Unix y Windows, así como los mecanismos para utilizar y entender los sistemas de medidas de almacenamiento.