



**CORPORACIÓN
UNIVERSITARIA
REMINGTON**

ESCUELA DE CIENCIAS BÁSICAS E INGENIERIA

ASIGNATURA: Sistemas Operativos

CORPORACIÓN UNIVERSITARIA REMINGTON
DIRECCIÓN PEDAGÓGICA

Este material es propiedad de la Corporación Universitaria Remington (CUR), para los estudiantes de la CUR en todo el país.

2011

CRÉDITOS



El módulo de estudio de la asignatura Sistemas Operativos es propiedad de la Corporación Universitaria Remington. Las imágenes fueron tomadas de diferentes fuentes que se relacionan en los derechos de autor y las citas en la bibliografía. El contenido del módulo está protegido por las leyes de derechos de autor que rigen al país.

Este material tiene fines educativos y no puede usarse con propósitos económicos o comerciales.

AUTOR

Javier Ospina Moreno

Tecnólogo en Sistemas de la Corporación Universitaria Remington Ingeniero de Sistemas de la Corporación Universitaria Remington Diplomado en Diseño Curricular. Corporación Universitaria Remington, Diplomado en Pedagogía para profesionales docentes (MEN) Universidad cooperativa Especialista en Sistemas. Corporación Universitaria Remington, Docente de la organización Remington Docente de Actuar famiempresas actualmente interactuar Docente de Politécnico Jaime Isaza Cadavid Docente de Tecnológico de Antioquia Docente de Tecnología e informática y emprendimiento en educación básica y media en la Institución Educativa Ángela Restrepo Moreno

Javier.ospina@remington.edu.co

Nota: el autor certificó (de manera verbal o escrita) No haber incurrido en fraude científico, plagio o vicios de autoría; en caso contrario eximió de toda responsabilidad a la Corporación Universitaria Remington, y se declaró como el único responsable.

RESPONSABLES

Dr. Mauricio Sepúlveda

Director de la Escuela Ciencias Básicas e Ingeniería

Director Pedagógico

Octavio Toro Chica

dirpedagogica.director@remington.edu.co

Coordinadora de Medios y Mediaciones

Angélica Ricaurte Avendaño

mediaciones.coordinador01@remington.edu.co

GRUPO DE APOYO

Personal de la Unidad de Medios y Mediaciones

EDICIÓN Y MONTAJE

Primera versión. Febrero de 2011.

Derechos Reservados



Esta obra es publicada bajo la licencia Creative Commons. Reconocimiento-No Comercial-Compartir Igual 2.5 Colombia.

TABLA DE CONTENIDO

| | | |
|-----------|---|-----------|
| 1. | MAPA DE LA ASIGNATURA | 8 |
| 2. | CONOCIMIENTO DEL HARDWARE EL SISTEMA OPERATIVO Y SU RELACIÓN | 10 |
| 2.1. | Definición de Sistema Operativo a Partir de sus Características | 12 |
| 2.2. | Conceptos Fundamentales Sobre Sistema Operativo | 15 |
| 2.2.1. | ¿Que es un proceso? | 15 |
| 2.3. | Clases y tipos de Sistemas Operativos | 16 |
| 2.3.1. | Sistemas Operativos por su Estructura | 16 |
| 2.3.2. | Máquina Virtual..... | 19 |
| 2.3.3. | Seguridad en Los Sistemas Operativos..... | 24 |
| 2.4. | Ejercicios por temas | 28 |
| 3. | GESTIÓN DE PROCESOS..... | 31 |
| 3.1. | Introducción a Los Procesos..... | 32 |
| 3.1.1. | Comunicación Entre Procesos | 43 |
| 3.1.2. | Dormir y despertar | 49 |
| 3.1.3. | El problema de productor-consumidor con transferencia de mensajes | 56 |
| 3.2. | Ejercicios por temas | 69 |
| 3.2.1. | Actividad..... | 71 |
| 4. | ADMINISTRADOR DE MEMORIA..... | 72 |
| 4.1. | Administración de la memoria sin intercambio o paginación..... | 73 |
| 4.1.1. | Administración Básica de Memoria | 73 |
| 4.2. | Intercambio | 77 |
| 4.2.1. | Administración de memoria con mapas de bits..... | 80 |
| 4.3. | Memoria virtual..... | 84 |
| 4.3.1. | Paginación | 84 |
| 4.3.2. | Algoritmos De Sustitución De Páginas | 90 |
| 4.4. | Ejercicios por temas | 95 |
| 5. | INTRODUCCIÓN A LOS SISTEMAS OPERATIVOS DISTRIBUIDOS | 96 |
| 5.1. | Sistemas Distribuidos | 96 |

| | | |
|------|----------------------------|-----|
| 5.2. | Ejercicios por temas | 105 |
| 5.3. | Glosario | 107 |
| 5.4. | Bibliografía | 113 |

1. MAPA DE LA ASIGNATURA

SISTEMAS OPERATIVOS

PROPÓSITO GENERAL DEL MÓDULO

Muestra la relación entre el Software y el hardware, generando un acercamiento al conocimiento y utilización de los Sistemas operativos que proporcionan las plataformas de uso diario en las empresas y hogares para el manejo de diferentes aplicaciones con las que solucionamos los distintos problemas relacionados con la tecnología informática de acuerdo a las distintas necesidades del hombre. Por ejemplo los programadores se basan en la plataforma para desarrollar sus programas y solucionar diversos problemas reales, los diseñadores utilizan las posibilidades que entrega la plataforma para crear sus diseños de todo tipo. Como los lenguajes de programación, circuitos digitales, diseño de compiladores y bases de datos relacionales.

También se pretende desarrollar en el estudiantes capacidades para plantear y resolver tablas de verdad, desarrollo conceptual de la teoría de conjuntos, funciones y relaciones, aritmética binaria y compuertas lógicas y su la relación de todo lo anterior con la tecnología e Ingeniería de sistemas.

OBJETIVO GENERAL

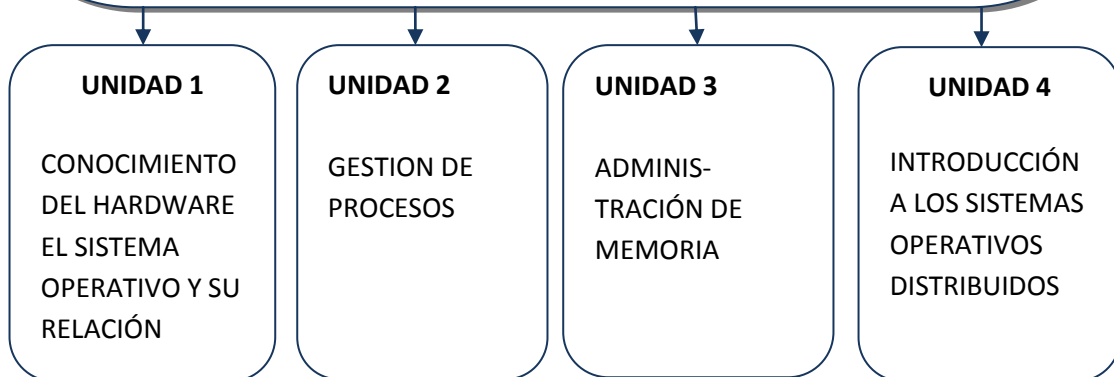
Desarrollar destrezas en el manejo a nivel administrativo de los últimos Sistemas Operativos(de Windows o de Linux) apoyados en el conocimiento profundo de su funcionamiento y en una vista practica de su uso más importante, a fin de permitir al estudiante enfrentarse a los problemas más apremiantes que el medio genera en esta materia, que le permiten a un estudiante de tecnología resolver de forma analítica y proyectiva las dificultades que en su cotidianidad se presentan sobre las distintas plataformas y sus usos.

OBJETIVOS ESPECÍFICOS

- ◆ Proporcionar la terminología suficiente que de una cultura mínima para entender el desarrollo del modulo
- ◆ Conocer y comprender la forma en que el sistema operativo administra la ejecución de procesos asociados a los diferentes programas de usuario y a la carga de tareas de núcleo en la memoria Ram.
- ◆ Conocer cómo se lleva el registro de memoria, para determinar la administración de su espacio cuando se hace ubicación de los procesos que se cargan o salida de procesos que terminan y optimizar su utilización.
- ◆ Conocer características generales de los sistemas operativos actuales que permitan facilitar su uso y aplicación

Estudiar el modelo lineal, de gran aplicación en todas las áreas del conocimiento. Los modelos matemáticos permiten la representación de situaciones problemáticas mediante el lenguaje matemático, facilitando de esta manera la manipulación matemática, soluciones generales y no particulares y su representación gráfica.

Capacitar al estudiante para usar los conceptos de la lógica matemática en las aplicaciones típicas de la tecnología informática e ingeniería de sistemas.



2. CONOCIMIENTO DEL HARDWARE EL SISTEMA OPERATIVO Y SU RELACIÓN

- Término es una
- expresión algebraica
- que consta de un solo
- símbolo o de varios
- símbolos no separados
- entre si por un signo.

<http://www.youtube.com/watch?v=zut8H1BaoFU&feature=related>

<http://www.youtube.com/watch?v=QEBqnFS5Mi4&feature=fvwrrel>

<http://www.youtube.com/watch?v=NYz6PEEdY4M&feature=related>

OBJETIVO GENERAL

- ◆ Proporcionar la terminología suficiente que de una cultura mínima para entender el desarrollo del módulo.

OBJETIVOS ESPECÍFICOS

- ◆ Comprender el concepto de Sistema Operativo y su aplicación
- ◆ Conocer la relación que hay entre el hardware y el Sistema Operativo
- ◆ Conocer sobre los sistemas operativos distribuidos y su aplicación actual

Prueba Inicial

Seleccione la respuesta correcta según su conocimiento previo:

Un sistema operativo permite:

- a. Solo administrar recursos
- b. Administra, controla dispositivos y permite ejecutar programas
- c. Controla dispositivos
- d. Permite solo la ejecución de programas.

El concepto principal para el sistema operativo es:

- a. Archivo
- b. Proceso
- c. Gestión administrativa
- d. Control de dispositivos

Un controlador de dispositivos se asocia al sistema operativo por:

- a. Permite la instalación del programa.
- b. Permite la gestión del recurso
- c. Permite la revisión del proceso
- d. Permite el control del dispositivo desde el sistema operativo.

La principal diferencia entre los sistemas operativos actuales es:

- a. Su sistema de archivos y sus funciones básicas.
- b. Solamente las opciones de red
- c. La administración de impresión
- d. El manejo de controladores.

2.1. Definición de Sistema Operativo a Partir de sus Características

El Sistema Operativo es un software que permite administrar los recursos de un computador: memoria, procesadores (llamados también microprocesadores), controlar los dispositivos físicos de la computadora (impresoras, teclados, mouses, tarjetas de video sonido y de red, entre otras; dispositivos de comunicación datos y dispositivos multimedia. También se considera como una plataforma sobre la cual se montan o instalan y corren las demás aplicaciones ejemplo Encarta, swit de office, graficadores, lenguajes de programación, manejadores de bases de datos, programas de audio y video, etc.

Los Sistemas Operativos se definen también como una máquina virtual, es decir, se presenta al usuario (el que trabaja con la computadora, puede ser en una estación de trabajo), para que sea más fácil la programación y como un controlador de los recursos, la labor del sistema operativo es la de proporcionar el control de los procesadores, memorias y dispositivos de Entrada y Salida, como el teclado, impresora, mouse, monitor, discos duros, y discos flexibles, etc.

“Un S. O. es un grupo de programas de proceso con las rutinas de control necesarias para mantener continuamente operativos dichos programas. El objetivo primario de un Sistema Operativo es Optimizar todos los recursos del sistema para soportar los requerimientos” ira a (<http://exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos/SO0.htm>).

Los sistemas operativos realizan muchas funciones, como proporcionar una interfaz amena y entendible para el usuario, para permitir aprovechar el hardware y los datos, evitando que se interfieran recíprocamente, planificar la distribución de los recursos de cómputo (como: impresora, discos duros, discos flexibles, mouse, teclado, servidores, scanners, etc. y el software, en donde podremos acceder a la comunicación con los datos) entre los usuarios , facilitar la entrada y salida de estos recursos, contabilizar el uso de estos recursos, recuperarse de los errores, facilitar las operaciones distribuidas (es decir, la transmisión de información cuando dos computadoras ó más están interconectadas a través de una red de datos) y operaciones en paralelo (es decir, cuando en una computadora se realizan dos procesos a la misma vez, por ejemplo, en el sistema operativo Windows 95, puedo obtener información de Internet y a la misma vez puedo pasar esta información a un procesador de textos como el Word.

Otro ejemplo podría ser el de un almacén de cadena cuando se registran la mercadería del cliente a través de cajas registradoras la cuales trabajan como computadoras ya que son unidades inteligentes que se encargan de registrar el precio, código, cantidad del artículo desde una misma base de datos, es decir son puntos de venta), organizar los datos para lograr un acceso rápido y

seguro, y manejar las comunicaciones en una red de computadoras. "Recuerde siempre para usar una aplicación en una computadora, primero debe cargarse el sistema operativo.

“El sistema operativo (a veces también citado mediante su forma abreviada OS en inglés) se encarga de crear el vínculo entre los recursos materiales, el usuario y las aplicaciones (procesador de texto, videojuegos, etcétera). Cuando un programa desea acceder a un recurso material, no necesita enviar información específica a los dispositivos periféricos; simplemente envía la información al sistema operativo, el cual la transmite a los periféricos correspondientes a través de su driver (controlador). Si no existe ningún driver, cada programa debe reconocer y tener presente la comunicación con cada tipo de periférico” ir a (http://es.wikibooks.org/wiki/Sistemas_operativos/Clasificaci%C3%B3n).

◆ **Administrador De Recursos**

| | |
|-----------|--|
| CPU | Procesador (Permite la ejecución de procesos) |
| RAM | Memoria Principal (permite almacenar temporalmente los procesos en RAM) |
| DISCO | Administración de Discos (espacio de almacenamiento permanente archivos en el DD) |
| IMPRESIÓN | Administración de la impresión (colas de impresión) |
| REDES | Comunicación (protocolos) Ej.: Directorio activo en Windows 2003 que permite la administración de(cpu,ram,disco,impresión) |

Considerando el Sistema Operativo como un administrador y controlador de recursos del computador, vemos las diversas funciones que el administrador debe hacer:

◆ **Funciones de la administración de memoria**

- ◆ Llevar registro del recurso (memoria). ¿Qué componentes se están usando y quién las usa?
- ◆ Si se esta ejecutando varios procesos, decidir el proceso que obtiene el control de la memoria, cuándo y cuánto.
- ◆ Asignar el recurso (memoria) cuando los procesos la solicitan.
- ◆ Recuperar el recurso (memoria) cuando el proceso ya no lo necesita o ha sido abortado.

◆ **Funciones de Administración del Procesador**

- ◆ Llevar el control del registro (procesadores y estado de los procesos)
- ◆ Decidir quién tendrá la oportunidad de utilizar el procesador.
- ◆ Asignar el recurso (procesador) a un proceso posicionando los registros necesarios de hardware.

- ◆ Recuperar el recurso (procesador) cuando el proceso cede el uso del programador, aborta o excede la cantidad permisible de utilización.

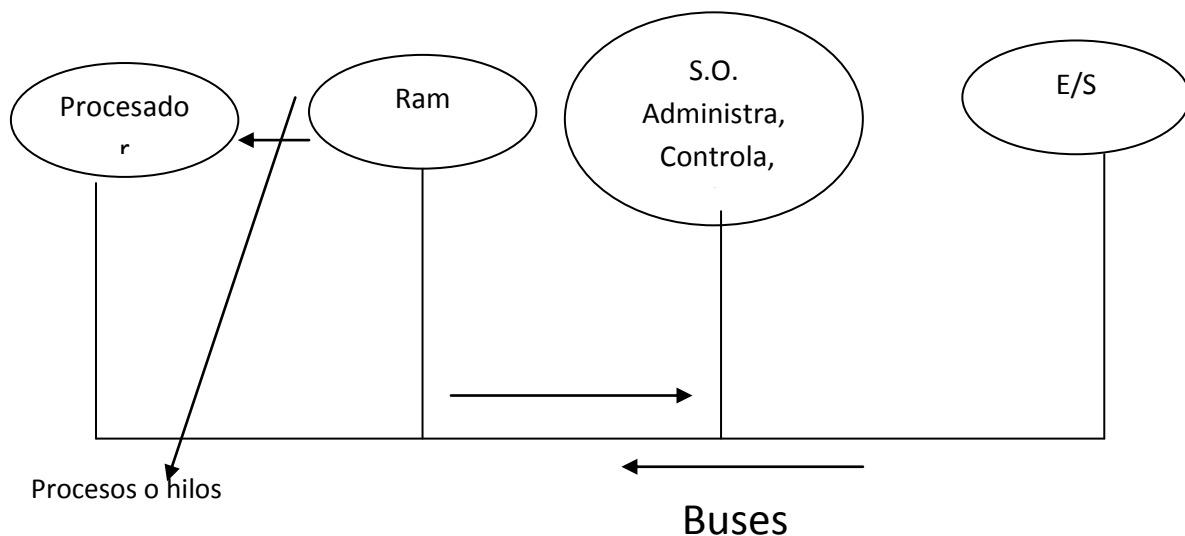
◆ **Funciones de la administración de los dispositivos**

- ◆ Mantener el control del recurso (dispositivo, canales y unidades de control).
- ◆ Decidir cuál es la forma eficiente de asignar el recurso (dispositivo). En caso de que deba compartirse, decidir quien lo recibe y cuanto recibirá, a esto se le llama planeación de entrada/salida.
- ◆ Asignar el recurso (dispositivo) y arrancar la operación de entrada/salida.
- ◆ Recuperar el recurso. En la mayoría de los casos, la entrada/salida termina automáticamente.

◆ **Funciones de la administración de información**

- ◆ Mantener el control del recurso (información), su posición, uso, estado, etc. Frecuentemente a estas facilidades se les llama el sistema de archivos.
- ◆ Decidir quién utiliza los recursos, imponer los requerimientos de protección y proporcionar rutinas de acceso.
- ◆ Asignar el recurso (información), por ejemplo cuando se abre un archivo.
- ◆ Desasignar el recurso, por ejemplo cuando se cierra un archivo.

◆ **Gráfico que muestra la relación entre software y Hardware a partir del Sistema Operativo**



2.2. Conceptos Fundamentales Sobre Sistema Operativo

2.2.1. ¿Que es un proceso?

Un proceso es básicamente un programa en ejecución. Esta compuesto del programa ejecutable, sus datos, además de toda la información necesaria para ejecutar el programa.

El sistema operativo decide detener la ejecución de un proceso y comenzar la ejecución de otro proceso.

Cuando un proceso se detiene en forma temporal, este debe volverse a inicializar en el mismo estado en que se encontraba al detenerse.

En muchos sistemas operativos, toda la información relativa a un proceso distinta del contenido de su propio espacio de dirección, se almacena en una tabla del sistema operativo llamada tablas de procesos, la cual consta de un arreglo (lista) de estructuras, una para cada proceso existente en ese momento.

Las llamadas al sistema de control de procesos fundamentales son las que se ocupan de la creación y fin de los procesos. Un proceso llamado intérprete de comando o shell lee los comando a partir de una computadora terminal.

2.2.1.1 El Núcleo y Los Procesos

El núcleo (también llamado kernel) de un sistema operativo es un conjunto de rutinas cuya misión es la de administrar el procesador, la memoria, la entrada/salida y el resto de recursos disponibles en la instalación. Toda esa administración la realiza para atender al funcionamiento y peticiones de los trabajos que se ejecutan en el sistema.

¿QUE ES UNA LLAMADA AL SISTEMA?

Una llamada al sistema se refiere a las solicitudes que requiere el usuario a través del hardware por medio del sistema operativo.

Los programas que tiene el usuario se comunican con el sistema operativo, estos programas le solicitan servicio mediante las llamadas al sistema.

El efecto de llamar a un procedimiento del programa consiste en que los datos del archivo dado pasen al almacén (buffer, que es un espacio de memoria) donde el programa pueda tener acceso a

ellos. Los programas deben siempre verificar los resultados de las llamadas al sistema para ver si no ocurren interrupciones.

El número y tipo de llamadas al sistema varía de un sistema operativo a otro. Por lo general, hay llamadas al sistema para crear procesos, controlar la memoria, leer y escribir y hacer labores de entrada/salida, como por ejemplo lectura de un terminal o impresión mediante una impresora.

2.3. Clases y tipos de Sistemas Operativos

Dar a conocer como se clasifican y subdividen los sistemas operativos de acuerdo al número de usuarios y tareas que se pueden manejar en ellos.

En esta sección se describirán las características que clasifican a los sistemas operativos, básicamente se cubrirán tres clasificaciones: sistemas operativos por su estructura (visión interna), sistemas operativos por los servicios que ofrecen y, finalmente, sistemas operativos por la forma en que ofrecen sus servicios (visión externa).

2.3.1. Sistemas Operativos por su Estructura

Según [Alcal92], se deben observar dos tipos de requisitos cuando se construye un sistema operativo, los cuales son:

Requisitos de usuario: Sistema fácil de usar y de aprender, seguro, rápido y adecuado al uso al que se le quiere destinar.

Requisitos del software: Donde se engloban aspectos como el mantenimiento, forma de operación, restricciones de uso, eficiencia, tolerancia frente a los errores y flexibilidad.

A continuación se describen las distintas estructuras que presentan los actuales sistemas operativos para satisfacer las necesidades que de ellos se quieren obtener.

2.3.1.1 Estructura monolítica

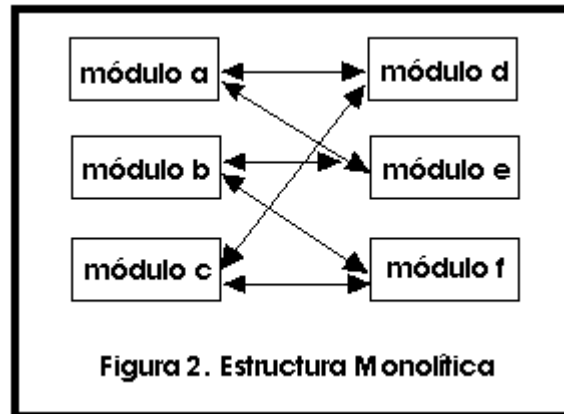
Es la estructura de los primeros sistemas operativos constituidos fundamentalmente por un solo programa compuesto de un conjunto de rutinas entrelazadas de tal forma que cada una puede llamar a cualquier otra (Ver Fig. 2). Las características fundamentales de este tipo de estructura son:

Construcción del programa final a base de módulos compilados separadamente que se unen a través del ligador.

Buena definición de parámetros de enlace entre las distintas rutinas existentes, que puede provocar mucho acoplamiento.

Carecen de protecciones y privilegios al entrar a rutinas que manejan diferentes aspectos de los recursos de la computadora, como memoria, disco, etc.

Generalmente están hechos a medida, por lo que son eficientes y rápidos en su ejecución y gestión, pero por lo mismo carecen de flexibilidad para soportar diferentes ambientes de trabajo o tipos de aplicaciones.

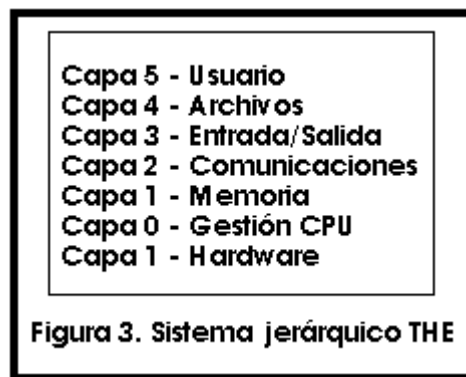


2.3.1.2 Estructura jerárquica

A medida que fueron creciendo las necesidades de los usuarios y se perfeccionaron los sistemas, se hizo necesaria una mayor organización del software, del sistema operativo, donde una parte del sistema contenía subpartes y esto organizado en forma de niveles.

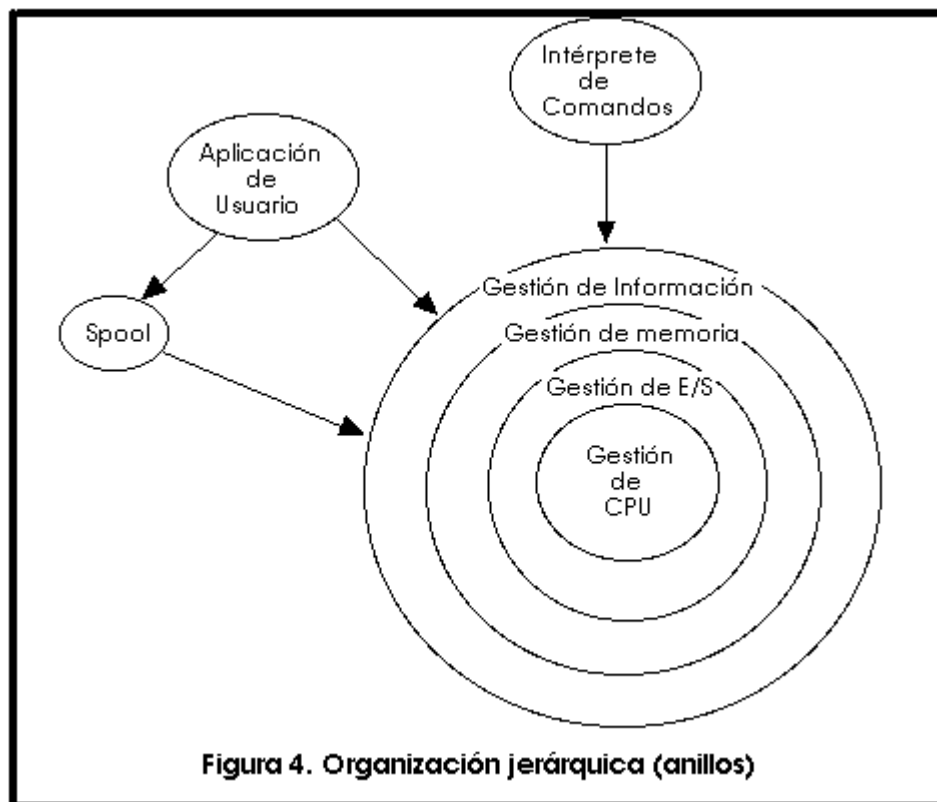
Se dividió el sistema operativo en pequeñas partes, de tal forma que cada una de ellas estuviera perfectamente definida y con un claro interface con el resto de elementos.

Se constituyó una estructura jerárquica o de niveles en los sistemas operativos, el primero de los cuales fue denominado THE (Technische Hogeschool, Eindhoven), de Dijkstra, que se utilizó con fines didácticos (Ver Fig. 3). Se puede pensar también en estos sistemas como si fueran 'multicapa'. Multics y Unix caen en esa categoría. [Feld93].



En la estructura anterior se basan prácticamente la mayoría de los sistemas operativos actuales. Otra forma de ver este tipo de sistema es la denominada de anillos concéntricos o "rings

“(Ver Fig. 4).

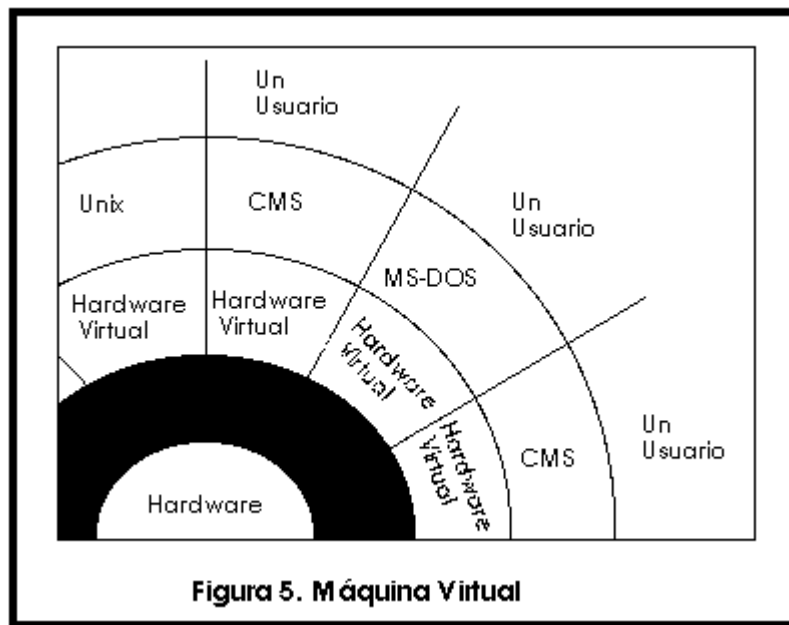


En el sistema de anillos, cada uno tiene una apertura, conocida como puerta o trampa (trap), por donde pueden entrar las llamadas de las capas inferiores. De esta forma, las zonas más internas del sistema operativo o núcleo del sistema estarán más protegidas de accesos indeseados desde las capas más externas. Las capas más internas serán, por tanto, más privilegiadas que las externas.

2.3.2. Máquina Virtual.

Se trata de un tipo de sistemas operativos que presentan una interface a cada proceso, mostrando una máquina que parece idéntica a la máquina real subyacente. Estos sistemas operativos separan dos conceptos que suelen estar unidos en el resto de sistemas: la multiprogramación y la máquina extendida. El objetivo de los sistemas operativos de máquina virtual es el de integrar distintos sistemas operativos dando la sensación de ser varias máquinas diferentes.

El núcleo de estos sistemas operativos se denomina monitor virtual y tiene como misión llevar a cabo la multiprogramación, presentando a los niveles superiores tantas máquinas virtuales como se soliciten. Estas máquinas virtuales no son máquinas extendidas, sino una réplica de la máquina real, de manera que en cada una de ellas se pueda ejecutar un sistema operativo diferente, que será el que ofrezca la máquina extendida al usuario (Ver Fig. 5).



2.3.2.1 Cliente-servidor (Microkernel)

El tipo más reciente de sistemas operativos es el denominado Cliente-servidor, que puede ser ejecutado en la mayoría de las computadoras, ya sean grandes o pequeñas.

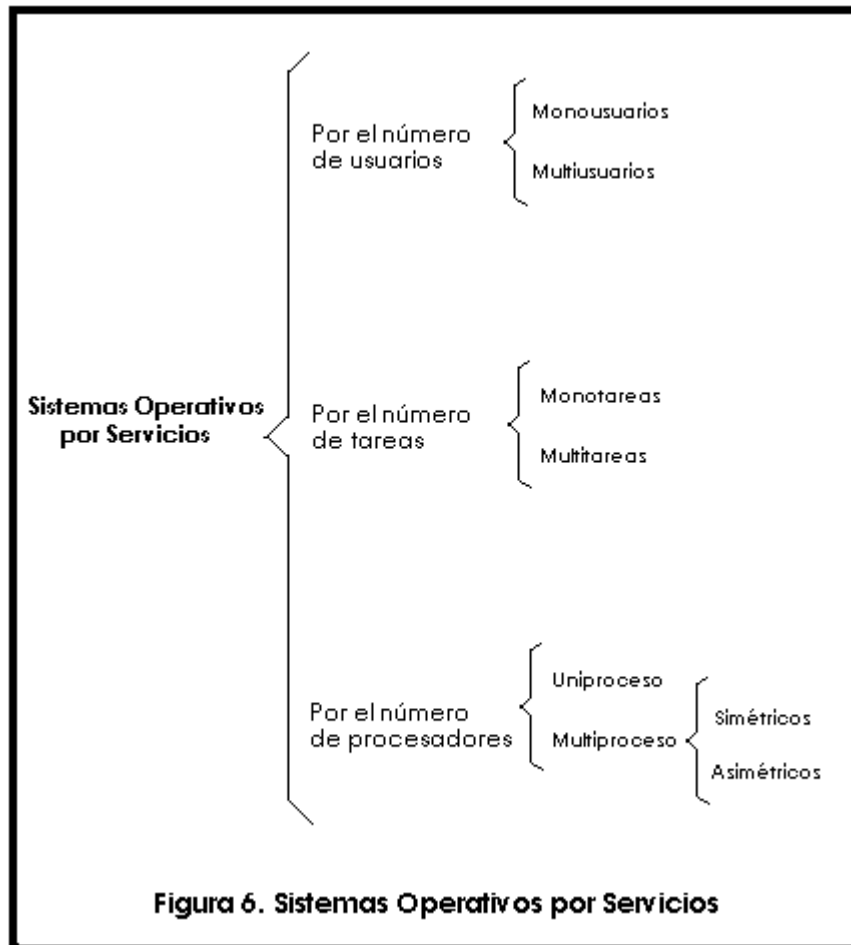
Este sistema sirve para toda clase de aplicaciones por tanto, es de propósito general y cumple con las mismas actividades que los sistemas operativos convencionales.

El núcleo tiene como misión establecer la comunicación entre los clientes y los servidores. Los procesos pueden ser tanto servidores como clientes. Por ejemplo, un programa de aplicación normal es un cliente que llama al servidor correspondiente para acceder a un archivo o realizar una operación de entrada/salida sobre un dispositivo concreto.

A su vez, un proceso cliente puede actuar como servidor para otro." [Alcal92]. Este paradigma ofrece gran flexibilidad en cuanto a los servicios posibles en el sistema final, ya que el núcleo provee solamente funciones muy básicas de memoria, entrada/salida, archivos y procesos, dejando a los servidores proveer la mayoría que el usuario final o programador puede usar. Estos servidores deben tener mecanismos de seguridad y protección que, a su vez, serán filtrados por el núcleo que controla el hardware. Actualmente se está trabajando en una versión de UNIX que contempla en su diseño este paradigma.

2.3.2.2 Sistemas Operativos por Servicios

Esta clasificación es la más comúnmente usada y conocida desde el punto de vista del usuario final. Esta clasificación se comprende fácilmente con el cuadro sinóptico que a continuación se muestra en la Fig. 6.



◆ Monousuarios

Los sistemas operativos monousuarios son aquéllos que soportan a un usuario a la vez, sin importar el número de procesadores que tenga la computadora o el número de procesos o tareas que el usuario pueda ejecutar en un mismo instante de tiempo. Las computadoras personales típicamente se han clasificado en este renglón.

◆ **Multiusuarios**

Los sistemas operativos multiusuarios son capaces de dar servicio a más de un usuario a la vez, ya sea por medio de varias terminales conectadas a la computadora o por medio de sesiones remotas en una red de comunicaciones. No importa el número de procesadores en la máquina ni el número de procesos que cada usuario puede ejecutar simultáneamente.

◆ **Monotareas**

Los sistemas monotarea son aquellos que sólo permiten una tarea a la vez por usuario. Puede darse el caso de un sistema multiusuario y monotarea, en el cual se admiten varios usuarios al mismo tiempo pero cada uno de ellos puede estar haciendo solo una tarea a la vez.

◆ **Multitareas**

Un sistema operativo multitarea es aquél que le permite al usuario estar realizando varias labores al mismo tiempo. Por ejemplo, puede estar editando el código fuente de un programa durante su depuración mientras compila otro programa, a la vez que está recibiendo correo electrónico en un proceso en background. Es común encontrar en ellos interfaces gráficas orientadas al uso de menús y el ratón, lo cual permite un rápido intercambio entre las tareas para el usuario, mejorando su productividad.

◆ **Uniprosceso**

Un sistema operativo uniprosceso es aquél que es capaz de manejar solamente un procesador de la computadora, de manera que si la computadora tuviese más de uno le sería inútil. El ejemplo más típico de este tipo de sistemas es el DOS y MacOS.

◆ **Multiprosceso**

Un sistema operativo multiprosceso se refiere al número de procesadores del sistema, que es más de uno y éste es capaz de usarlos todos para distribuir su carga de trabajo. Generalmente estos sistemas trabajan de dos formas: simétrica o asimétricamente. Cuando se trabaja de manera asimétrica, el sistema operativo selecciona a uno de los procesadores el cual jugará el papel de procesador maestro y servirá como pivote para distribuir la carga a los demás procesadores, que reciben el nombre de esclavos. Cuando se trabaja de manera simétrica, los procesos o partes de ellos (threads) son enviados indistintamente a cualesquiera de los procesadores disponibles, teniendo, teóricamente, una mejor distribución y equilibrio en la carga de trabajo bajo este esquema.

Se dice que un thread es la parte activa en memoria y corriendo de un proceso, lo cual puede consistir de un área de memoria, un conjunto de registros con valores específicos, la pila y otros valores de contexto. Un aspecto importante a considerar en estos sistemas es la forma de crear aplicaciones para aprovechar los varios procesadores. Existen aplicaciones que fueron hechas para correr en sistemas monoproceso que no toman ninguna ventaja a menos que el sistema operativo

o el compilador detecte secciones de código paralelizables, los cuales son ejecutados al mismo tiempo en procesadores diferentes. Por otro lado, el programador puede modificar sus algoritmos y aprovechar por sí mismo esta facilidad, pero esta última opción las más de las veces es costosa en horas hombre y muy tediosa, obligando al programador a ocupar tanto o más tiempo a la paralelización que a elaborar el algoritmo inicial.

a) Sistemas Operativos por la Forma de Ofrecer sus Servicios

Esta clasificación también se refiere a una visión externa, que en este caso se refiere a la del usuario, el cómo accesa los servicios. Bajo esta clasificación se pueden detectar dos tipos principales: sistemas operativos de red y sistemas operativos distribuidos.

◆ Sistemas Operativos de Red

Los sistemas operativos de red se definen como aquellos que tienen la capacidad de interactuar con sistemas operativos en otras computadoras por medio de un medio de transmisión con el objeto de intercambiar información, transferir archivos, ejecutar comandos remotos y un sin fin de otras actividades. El punto crucial de estos sistemas es que el usuario debe saber la sintaxis de un conjunto de comandos o llamadas al sistema para ejecutar estas operaciones, además de la ubicación de los recursos que desee acceder. Por ejemplo, si un usuario en la computadora hidalgo necesita el archivo matriz.pas que se localiza en el directorio /software/código en la computadora morelos bajo el sistema operativo UNIX, dicho usuario podría copiarlo a través de la red con los comandos siguientes: hidalgo% hidalgo% rcp morelos:/software/código/matriz.pas. hidalgo% En este caso, el comando rcp que significa "remote copy" trae el archivo indicado de la computadora morelos y lo coloca en el directorio donde se ejecutó el mencionado comando. Lo importante es hacer ver que el usuario puede acceder y compartir muchos recursos.

◆ Sistemas Operativos Distribuidos

Los sistemas operativos distribuidos abarcan los servicios de los de red, logrando integrar recursos (impresoras, unidades de respaldo, memoria, procesos, unidades centrales de proceso) en una sola máquina virtual que el usuario accesa en forma transparente. Es decir, ahora el usuario ya no necesita saber la ubicación de los recursos, sino que los conoce por nombre y simplemente los usa como si todos ellos fuesen locales a su lugar de trabajo habitual.

Todo lo anterior es el marco teórico de lo que se desearía tener como sistema operativo distribuido, pero en la realidad no se ha conseguido crear uno del todo, por la complejidad que suponen: distribuir los procesos en las varias unidades de procesamiento, reintegrar sub-resultados, resolver problemas de concurrencia y paralelismo, recuperarse de fallas de algunos recursos distribuidos y consolidar la protección y seguridad entre los diferentes componentes del sistema y los usuarios. [Tan92]. Los avances tecnológicos en las redes de área local y la creación de microprocesadores de 32 y 64 bits lograron que computadoras más o menos baratas tuvieran el suficiente poder en forma autónoma para desafiar en cierto grado a los mainframes, y a la vez se

dio la posibilidad de intercomunicarlas, sugiriendo la oportunidad de partir procesos muy pesados en cálculo en unidades más pequeñas y distribuirlas en los varios microprocesadores para luego reunir los sub-resultados, creando así una máquina virtual en la red que exceda en poder a un mainframe. El sistema integrador de los microprocesadores que hacer ver a las varias memorias, procesadores, y todos los demás recursos como una sola entidad en forma transparente se le llama sistema operativo distribuido.

Las razones para crear o adoptar sistemas distribuidos se dan por dos razones principales: por necesidad (debido a que los problemas a resolver son inherentemente distribuidos) o porque se desea tener más confiabilidad y disponibilidad de recursos. En el primer caso tenemos, por ejemplo, el control de los cajeros automáticos en diferentes estados de la república. Ahí no es posible ni eficiente mantener un control centralizado, es más, no existe capacidad de cómputo y de entrada/salida para dar servicio a los millones de operaciones por minuto.

En el segundo caso, supóngase que se tienen en una gran empresa varios grupos de trabajo, cada uno necesita almacenar grandes cantidades de información en disco duro con una alta confiabilidad y disponibilidad. La solución puede ser que para cada grupo de trabajo se asigne una partición de disco duro en servidores diferentes, de manera que si uno de los servidores falla, no se deje dar el servicio a todos, sino sólo a unos cuantos y, más aún, se podría tener un sistema con discos en espejo (mirror) a través de la red, de manera que si un servidor se cae, el servidor en espejo continúa trabajando y el usuario ni cuenta se da de estas fallas, es decir, obtiene acceso a recursos en forma transparente.

2.3.3. Seguridad en Los Sistemas Operativos

En los sistemas operativos se requiere tener una buena seguridad informática, tanto del hardware, programas y datos, previamente haciendo un balance de los requerimientos y mecanismos necesarios. Con el fin de asegurar la integridad de la información contenida.

Dependiendo de los mecanismos utilizados y de su grado de efectividad, se puede hablar de sistemas seguros e inseguros. En primer lugar, deben imponerse ciertas características en el entorno donde se encuentra la instalación de los equipos, con el fin de impedir el acceso a personas no autorizadas, mantener un buen estado y uso del material y equipos, así como eliminar los riesgos de causas de fuerza mayor, que puedan destruir la instalación y la información contenida.

En la actualidad son muchas las violaciones que se producen en los sistemas informáticos, en general por acceso de personas no autorizadas que obtienen información confidencial pudiendo incluso manipularla. En ocasiones este tipo de incidencias resulta grave por la naturaleza de los datos; por ejemplo si se trata de datos bancarios, datos oficiales que puedan afectar a la seguridad de los estados, etc.

El software mal intencionado que se produce por diversas causas, es decir pequeños programas que poseen gran facilidad para reproducirse y ejecutarse, cuyos efectos son destructivos nos estamos refiriendo a los virus informáticos.

Para esto, se analizan cuestiones de seguridad desde dos perspectivas diferentes la seguridad externa y la seguridad interna.

Todos los mecanismos dirigidos a asegurar el sistema informático sin que el propio sistema intervenga en el mismo se engloban en lo que podemos denominar seguridad externa.

La seguridad externa puede dividirse en dos grandes grupos:

Seguridad física. Engloba aquellos mecanismos que impiden a los agentes físicos la destrucción de la información existente en el sistema; entre ellos podemos citar el fuego, el humo, inundaciones descargas eléctricas, campos magnéticos, acceso físico de personas con no muy buena intención, entre otros.

Seguridad de administración. Engloba los mecanismos más usuales para impedir el acceso lógico de personas físicas al sistema.

Todos los mecanismos dirigidos a asegurar el sistema informático, siendo el propio sistema el que controla dichos mecanismos, se engloban en lo que podemos denominar seguridad interna.

Sistemas Operativos Distribuidos

<http://www.augcyl.org/?q=glol-intro-sistemas-distribuidos>

Los sistemas operativos distribuidos desempeñan las mismas funciones que un sistema operativo normal, pero con la diferencia de trabajar en un entorno distribuido. Su Misión principal consiste en facilitar el acceso y la gestión de los recursos distribuidos en la red.

En un sistema operativo distribuido los usuarios pueden acceder a recursos remotos de la misma manera en que lo hacen para los recursos locales. Permiten distribuir trabajos, tareas o procesos, entre un conjunto de procesadores. Puede ser que este conjunto de procesadores esté en un equipo o en diferentes, lo cual es transparente para el usuario.

Características básicas

Los sistemas operativos distribuidos están basados en las ideas básicas:

- ◆ Transparencia
- ◆ Eficiencia
- ◆ Flexibilidad
- ◆ Transparencia

El concepto de transparencia de un Sistema operativo distribuido va ligado a la idea de que todo el sistema funcione de forma similar en todos los puntos de la red, debido a esto queda como labor del sistema operativo coordinar el mecanismo que logre la unificación de todos los sistemas y recursos totalmente transparente para el usuario o aplicación.

El que el sistema disponga de varios procesadores debe lograr un mayor rendimiento del sistema, pero el sistema operativo debe controlar que tanto los usuarios como los programadores vean el núcleo del sistema distribuido como un único procesador, Es decir que la programación y la ejecución de los programas y tareas sean exactamente iguales que las de los sistemas operativos normales en aspectos visuales y de programación, pero más rápidos y eficientes por la distribución de la tareas.

◆ **Eficiencia**

La idea base de los sistemas operativos distribuido es la de obtener sistemas mucho más rápidos que los utilizados de procesador único, Y para lograr esto tenemos que olvidar la idea antigua de ejecutar los programas en estos procesadores y pensar en distribuir las tareas a los procesadores libres más rápidos en cada momento.

El concepto global de que un procesador haga todas las tareas y la desarrolle rápido depende de muchos factores concretos: Velocidad, Memoria y tipo de procesamiento, Pero para un sistema operativo distribuido esto es mucho más fácil y eficiente, solo buscara un procesador más rápido y más libre para que desarrolle las tareas y hará un display de los resultados obtenidos.

◆ **Flexibilidad**

La Flexibilidad dentro de sistema operativo distribuido, describe su capacidad para soportar cambios, actualizaciones y mejoras que le permitan irse desarrollando al mismo ritmo de la evolución tecnológica.

Dicha capacidad es una virtud y un conflicto. Una Virtud debido a las grandes necesidades de los sistemas operativos de mejorar después de las primeras versiones y un conflicto que surge entre los sistemas de con Núcleo Monolítico y los sistemas con Micro núcleo las cuales son dos arquitecturas distintas del núcleo del sistema operativo.

◆ **Núcleo Monolítico**

Como ejemplo de sistema operativo de núcleo monolítico esta UNIX, estos sistemas tienen en núcleo grande y complejo, que engloba todos los servicios del sistema. Esta programado de forma no modular, y tiene un rendimiento mayor que un micro núcleo. Sin embargo, cualquier cambio a realzar en cualquiera de los servicios, requiere de hacer un STOP a todos los servicios y la recopilación del núcleo.

◆ **Micro Núcleo.**

La arquitectura ofrece la alternativa al núcleo monolítico, se basa en una programación altamente modular y tiene un tamaño mucho menor que el núcleo monolítico. Como consecuencia, el refinamiento y el control de errores son más rápidos y sencillos. Además, la actualización de los servicios es más sencilla y ágil. Ya que solo es necesario la recopilación del servicio y no de todo el núcleo. Como desventaja, El rendimiento se ve afectado negativamente.

En la actualidad la mayoría de los sistemas operativos distribuidos en desarrollo tienden a un diseño de micro núcleo el cual aun siendo un poco más lento, garantiza una estabilidad mayor y un aumento de la flexibilidad del sistema.

◆ **Escalabilidad**

Un sistema operativo distribuido debería funcionar tanto para una docena de computadoras como para mil en una sola red, el tipo de red utilizada no debe de ser un problema ni su topología (LAN o WAN) (TOKEN RING o ETHERNET) y mucho menos la distancia entre los equipos. Sin embargo todo esto influye, Aunque estos puntos serían muy deseables, puede que la solución válida para unas cuantas computadoras no sean aplicables como para mil. Del mismo modo el tipo de red condiciona grandemente el rendimiento del sistema y puede que lo funcione para un tipo de red requiera modificaciones para otro.

Los sistemas operativos distribuidos necesitan de grandes estándares para trabajar y sobre todo de ajustes a las necesidades principales de cada red y sus usuarios. Este concepto propone que cualquier computador debe funcionar perfectamente como un sistema operativo distribuido, pero de la misma forma debe de formar parte y trabajar como más equipos no importan la cantidad o los recursos que estos le puedan proporcionar.

◆ **Sincronización**

La sincronización es un punto clave para los sistemas operativos distribuidos. Para computadores únicos no es nada importante, pero en el caso de los recursos compartidos de la red, la sincronización es sumamente importante.

Los sistemas operativos distribuidos tienen un reloj por cada ordenador del sistema, con lo que es fundamental una coordinación entre todos los relojes para mostrar una hora única. Los osciladores de cada ordenador son ligeramente diferentes, y como consecuencia todo los relojes sufren un desfase y deben ser sincronizados continuamente. La sincronización no es trivial, porque se realiza a través de mensajes por la red. Cuyo tiempo de envío puede ser variable y depender de muchos factores como la distancia, la velocidad de transmisión y la propia estructura de la red.

◆ **El Reloj.**

La sincronización del reloj no tiene que ser exacta y bastara con que sea aproximadamente igual en todos los ordenadores. Hay que tener en cuenta eso sí. El modo de actualizar la hora de un reloj es particular. Es fundamenta no retrasar nunca la hora, aunque el reloj adelante. En vez de eso, hay que atrasar la actualizaron del reloj. Frenarlo. Hasta que alcance la hora aproximada. Existen diferentes algoritmos de actualizan de la hora.

El Reloj es únicamente uno de los tantos problemas de sincronización que existen en los sistemas operativos distribuidos.

◆ **Desventajas de los SOD**

Por muy maravillosos que nos puedan pareces los sistemas operativos distribuidos, también tienen sus desventajas. La sincronización del sistema es una tarea ardua de la cual nunca se descansa y la estandarización del sistema es un tanto complicada y limitante.

Debido a que no todos los sistemas operativos son de carácter distribuido enlazar los distintos tipos de sistemas operativos es un poco complicado.

El interés de hacer el SOD lo más transparente posible lo hace muy complicado en su programación y el lograr que el sistema operativo no tenga problemas para que no cause problemas a otros equipos que le asignaron tareas es un poco dificultoso. ir a <http://es.kioskea.net/contents/systemes/sysintro.php3>

2.4. Ejercicios por temas

◆ **Ejercicio tema 1.**

1. Escriba 5 ejemplos específicos en los cuales el sistema controla el Hardware
2. Como se entiende la siguiente frase: “El sistema operativo administra el almacenamiento y el procesamiento”
3. Una plataforma que permite la ejecución de aplicaciones que parte del sistema operativo explica.

◆ **Ejercicio tema 2**

1. Los procesos de núcleo en el sistema operativo se usan para:
 - a. Ejecución de tareas normales de usuario
 - b. Solo procesos o hilos que manejan la memoria
 - c. El control de todo tipo de tareas administrativas del sistema operativo

- d. Procesos fundamentales de control y gestión

2. La multiprogramación es seudoparalela por:

- a. Permite la ejecución de dos o más tareas al mismo tiempo
- b. Permite hacer parecer que la ejecución se da al mismo tiempo
- c. Permite que dos usuarios puedan trabajar con el mismo proceso
- d. Permite que con dos procesadores se atiendan dos tareas al tiempo

3. Una llamada al sistema se usa para :

- a. Ejecutar un proceso de usuario
- b. Ejecutar más de un proceso de usuario a la vez
- c. Terminar un proceso de Usuario
- d. Crear objetos del sistema como procesos o archivos

◆ **Ejercicio tema 3**

1. En una tabla como la que se visualiza a continuación escribir los nombres de los diferentes sistemas operativos que han existido y existen actualmente y clasificarlos de acuerdo a lo que se pide en las demás columnas

| Sistema operativo | por su Estructura | por Servicios | por la Forma de Ofrecer sus Servicios |
|-------------------|-------------------|---------------|---------------------------------------|
| | | | |
| | | | |
| | | | |
| | | | |

Prueba Final

1. En qué clase de sistemas se dan la multiprogramación y el multiproceso. ¿Se pueden dar juntos o no?. Diga claramente cuál sería la relación.
2. Explica que diferencias puedes encontrar entre: El modelo cliente-servidor-Un esquema de red centralizada y una Distribuida. Intenta dar algún ejemplo que clarifique las diferencias.
- 3.Cuál es la relación entre la capa de bajo nivel y los compiladores, editores e intérpretes de comandos y el sistema operativo.
4. Con cuál de las características de un sistema distribuido se relacionan los siguientes casos reales. Explique adicionalmente.
 - a. En Bancolombia un disco duro reemplazo a otro que sufrió un daño por 2 horas
 - b. A Un intruso se le impidió ingresar al sistema de Isa.
 - c. La nueva sala de EPM tendrá 16 computadoras de última generación pero podrá con 16 computadoras más dentro de la red.
 - d. La respuesta de procesamiento de los servidores de Isa mejoro en un 15%.
 - e. Con el nuevo programa de apoyo a las transacciones del banco de Bogotá se mejora el servicio a los clientes y la preparación de informes.
1. explique porque el S.O. es una máquina virtual y extendida

3. GESTIÓN DE PROCESOS

OBJETIVO GENERAL

Comprender la forma en que el sistema operativo administra la ejecución de procesos asociados a los diferentes programas de usuario y a la carga de tareas de núcleo en la memoria Ram.

OBJETIVOS ESPECÍFICOS

Estudiar la planificación de procesos asociados al procesador
Conceptuar sobre los diferentes algoritmos de planificación de procesos
Estudiar el problema de concurrencia en la ejecución de procesos

Prueba Inicial

Un programa escrito para solucionar un problema es un proceso cuando:

- a. Cuando fue codificado en un lenguaje
- b. Cuando fueron corregidos sus errores de compilación
- c. Cuando se almacena como un archivo en el disco duro
- d. Cuando se ejecuta en la computadora

La comunicación de un proceso con las partes básicas de la computadora se hace a través de:

- a. Los dispositivos de entrada y salida
- b. El disco duro
- c. Los buses de transferencia de datos
- d. La memoria principal

La diferencia entre los procesos de núcleo y los procesos de un sistema operativo es:

- a. El peso de los procesos en ram
- b. El lugar que ocupan en la memoria ram
- c. La forma en que se ejecutan con el procesador
- d. La planificación entre ellos

La variable de estado indica para un proceso lo siguiente:

- a. La forma como se encuentra el proceso en memoria

- b. Que el procesador tiene prioridad de ejecución sobre alguno de ellos
- c. Que los procesos pueden compartir memoria
- d. Que los procesos deben tener tiempos comunes de ejecución

La exclusión mutua entre procesos se define como:

- a. Que dos procesos se ejecutan en un área común de memoria en paralelo
- b. Que dos procesos se ejecutan en una memoria compartida pero no al mismo tiempo
- c. Que un proceso se ejecuta y posteriormente el otro
- d. Que no hay posibilidad de control del sistema operativo sobre las regiones críticas

3.1. Introducción a Los Procesos

<http://laurel.datsi.fi.upm.es/docencia/asignaturas/sod>

Andrew S. Tanenbaum: "Sistemas Operativos Distribuidos", Prentice Hall (1996).

Todas las computadoras modernas pueden hacer varias cosas al mismo tiempo. Mientras ejecuta un programa de usuario, una computadora también puede estar leyendo de un disco y enviando texto a una pantalla o impresora. En un sistema de multiprogramación, la CPU también conmuta de un programa a otro, ejecutando cada uno durante decenas o centenas de milisegundos. Si bien, estrictamente hablando, en un instante dado la CPU está ejecutando sólo un programa, en el curso de un segundo puede trabajar con varios programas, dando a los usuarios la ilusión de paralelismo.

A veces se usa el término seudoparalelismo para referirse a esta rápida conmutación de la CPU entre programas, para distinguirla del verdadero paralelismo de hardware de los sistemas multiprocesador (que tienen dos o más CPU que comparten la misma memoria física). Para el ser humano es difícil seguir la pista a múltiples actividades paralelas. Por ello, los diseñadores de sistemas operativos han desarrollado a lo largo de los años un modelo (procesos secuenciales) que facilita el manejo del paralelismo. Ese modelo y sus usos son el tema de este capítulo.

◆ El modelo de procesos

En este modelo, todo el software ejecutable de la computadora, lo que a menudo incluye al sistema operativo, está organizado en una serie de procesos secuenciales, o simplemente procesos. Un proceso no es más que un programa en ejecución, e incluye los valores actuales del contador de programa, los registros y las variables. Conceptualmente, cada uno de estos procesos tiene su propia CPU virtual. Desde luego, en la realidad la verdadera CPU conmuta de un proceso a otro, pero para entender el sistema es mucho más fácil pensar en una colección de procesos que se ejecutan en (seudo) paralelo que tratar de seguir la pista a la forma en que la CPU conmuta de

un programa a otro. Esta rápida conmutación se denomina multiprogramación, como vimos en el capítulo anterior.

En la Fig. 2-1(a) vemos una computadora multiprogramando dos programas en la memoria. En la Fig. 2-1(b) vemos cuatro procesos, cada uno con su propio flujo de control (esto es, su propio contador de programa), ejecutándose con independencia de los otros. En la Fig. 2-1(c) vemos que si el intervalo de tiempo es suficientemente largo, todos los procesos avanzan, pero en un instante dado sólo un proceso se está ejecutando realmente.

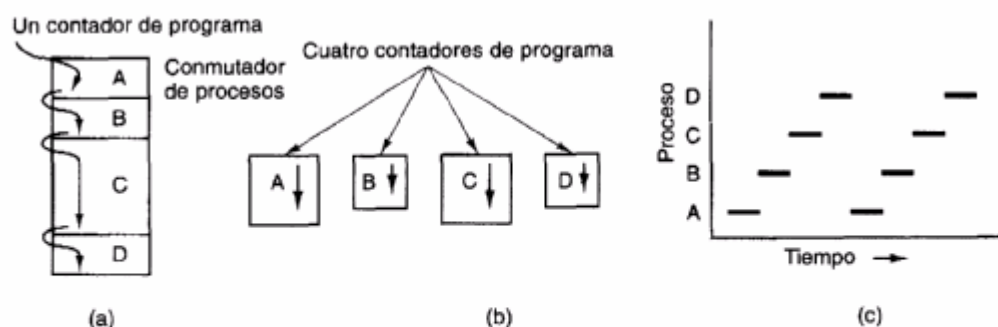


Figura 2-1. (a) Multiprogramación de cuatro programas. (b) Modelo conceptual de cuatro procesos secuenciales independientes. (c) Sólo un programa está activo en un instante dado.

Con la CPU conmutando entre los procesos, la rapidez con que un proceso realiza sus cálculos no es uniforme, y probablemente ni siquiera reproducible si los mismos procesos se ejecutan otra vez. Por tanto, los procesos no deben programarse basándose en supuestos acerca de los tiempos. Considere, por ejemplo, un proceso de E/S que inicia una cinta continua para restaurar archivos respaldados, ejecuta un ciclo vacío 10 000 veces para permitir que la cinta alcance la velocidad de trabajo y luego emite un comando para leer el primer registro.

Si la CPU decide conmutar a otro proceso durante el ciclo vacío, es posible que los procesos de cinta no se ejecuten otra vez sino hasta después de que el primer registro haya pasado por la cabeza de lectura. Cuando un proceso tiene requisitos de tiempo real críticos como éste, es decir, cuando ciertos eventos deben ocurrir en cierto número de milisegundos, se deben tomar medidas especiales para asegurar que así ocurran. Normalmente, empero, los procesos no resultan afectados por la multiprogramación subyacente de la CPU ni las velocidades relativas de los diferentes procesos.

La diferencia entre un programa y un proceso es sutil, pero crucial. Tal vez una analogía ayude a aclarar este punto. Consideremos un computólogo con inclinaciones gastronómicas que está preparando un pastel de cumpleaños para su hija. Él cuenta con una receta para pastel de cumpleaños y una cocina bien abastecida de las entradas necesarias: harina, huevos, azúcar,

extracto de vainilla, etc. En esta analogía, la receta es el programa (es decir, un algoritmo expresado en alguna notación apropiada), el computólogo es el procesador (CPU) y los ingredientes del pastel son los datos de entrada. El proceso es la actividad de nuestro pastelero consistente en leer la receta, obtener los ingredientes y hornear el pastel.

Imaginemos ahora que el hijo del computólogo llega corriendo y llorando, diciendo que le picó una abeja. El computólogo registra el punto en que estaba en la receta (guarda el estado del proceso actual), saca un libro de primeros auxilios, y comienza a seguir las instrucciones que contiene. Aquí vemos cómo el procesador se conmuta de un proceso (hornear) a un proceso de más alta prioridad (administrar cuidados médicos), cada uno con un programa diferente (receta vs. libro de primeros auxilios). Una vez que se ha atendido la picadura de abeja, el computólogo regresa a su pastel, continuando en el punto donde había interrumpido.

La idea clave aquí es que un proceso es una actividad de algún tipo: tiene programa, entrada, salida y un estado. Se puede compartir un procesador entre varios procesos, usando algún algoritmo de planificación para determinar cuándo debe dejarse de trabajar en un proceso para atender a uno distinto.

◆ Jerarquías de procesos

Los sistemas operativos que manejan el concepto de proceso deben contar con algún mecanismo para crear todos los procesos necesarios. En los sistemas muy sencillos, o en los diseñados para ejecutar sólo una aplicación (p. ej., controlar un dispositivo en tiempo real), es posible que, cuando el sistema se inicia, todos los procesos que puedan necesitarse estén presentes. Sin embargo, en la mayor parte de los sistemas se necesita algún mecanismo para crear y destruir procesos según sea necesario durante la operación.

En MINIX, los procesos se crean con la llamada al sistema FORK (bifurcar), que crea una copia idéntica del proceso invocador. El proceso hijo también puede ejecutar FORK, así que es posible tener un árbol de procesos. En otros sistemas operativos existen llamadas al sistema para crear un proceso, cargar su memoria y ponerlo a ejecutar. Sea cual sea la naturaleza exacta de la llamada al sistema, los procesos necesitan poder crear otros procesos. Observe que cada proceso tiene un padre, pero cero, uno, dos o más hijos.

Como ejemplo sencillo del uso de los árboles de procesos, veamos la forma en que MINIX se inicializa a sí mismo cuando se pone en marcha. Un proceso especial, llamado `init`, está presente en la imagen de arranque. Cuando este proceso comienza a ejecutarse, lee un archivo que le indica cuántas terminales hay, y luego bifurca un proceso nuevo por terminal. Estos procesos esperan a que alguien inicie una sesión. Si un inicio de sesión (login) tiene éxito, el proceso de inicio de sesión ejecuta un shell para aceptar comandos. Estos comandos pueden iniciar más procesos, y así sucesivamente. Por tanto, todos los procesos del sistema pertenecen a un mismo

árbol, que tiene a mit en su raíz. (El código de mit no se lista en este libro, y tampoco el del shell. Había que poner un límite en algún lado.)

◆ Estados de procesos

Aunque cada proceso es una entidad independiente, con su propio contador de programa y estado interno, los procesos a menudo necesitan interactuar con otros procesos. Un proceso podría generar ciertas salidas que otro proceso utiliza como entradas. En el comando de Shell `cat capítulo2 capítulo3 grep árbol` el primer proceso, que ejecuta `cat`, concatena y envía a la salida tres archivos. El segundo proceso, que ejecuta `grep`, selecciona todas las líneas que contienen la palabra “árbol”. Dependiendo de las velocidades relativas de los dos procesos (que a su vez dependen de la complejidad relativa de los programas y de cuánto tiempo de CPU ha ocupado cada uno), puede suceder que `grep` esté listo para ejecutarse, pero no haya entradas esperando ser procesadas por él. En tal caso, `grep` deberá bloquearse hasta que haya entradas disponibles.

Cuando un proceso se bloquea, lo hace porque le es imposible continuar lógicamente, casi siempre porque está esperando entradas que todavía no están disponibles. También puede ser que un programa que conceptualmente está listo y en condiciones de ejecutarse sea detenido porque el sistema operativo ha decidido asignar la CPU a otro proceso durante un tiempo. Estas dos condiciones son totalmente distintas. En el primer caso, la suspensión es inherente al problema (no es posible procesar la línea de comandos del usuario antes de que éste la teclee). En el segundo caso, se trata de un tecnicismo del sistema (no hay suficientes CPU para darle a cada proceso su propio procesador privado). En la Fig. 2-2 vemos un diagrama de estados que muestra los tres estados en los que un proceso puede estar:

1. Ejecutándose (usando realmente la CPU en ese instante).
2. Listo (se puede ejecutar, pero se suspendió temporalmente para dejar que otro proceso se ejecute).
3. Bloqueado (no puede ejecutarse en tanto no ocurra algún evento externo).

Lógicamente, los dos primeros estados son similares. En ambos casos el proceso está dispuesto a ejecutarse, sólo que en el segundo temporalmente no hay una CPU a su disposición. El tercer estado es diferente de los primeros dos en cuanto a que el proceso no puede ejecutarse, incluso si la CPU no tiene nada más que hacer.

Puede haber cuatro transiciones entre estos tres estados, como se muestra. La transición 1 ocurre cuando un proceso descubre que no puede continuar. En algunos sistemas el proceso debe ejecutar una llamada al sistema, `BLOCK`, para pasar al estado bloqueado. En otros sistemas,

incluido MINIX, cuando un proceso lee de un conducto o de un archivo especial (p. ej., una terminal) y no hay entradas disponibles, se bloquea automáticamente.

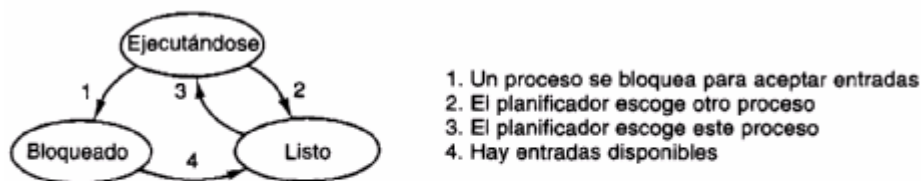


Figura 2-2. Un proceso puede estar en el estado de ejecutándose, bloqueado o listo. Las transiciones entre estos tres estados son las que se muestran.

Las transiciones 2 y 3 son causadas por el planificador de procesos, una parte del sistema operativo, sin que el proceso se entere siquiera de ellas. La transición 2 ocurre cuando el planificador decide que el proceso en ejecución ya se ejecutó durante suficiente tiempo y es hora de dejar que otros procesos tengan algo de tiempo de CPU. La transición 3 ocurre cuando todos los demás procesos han disfrutado de una porción justa y es hora de que el primer proceso reciba otra vez la CPU para ejecutarse. El tema de la planificación, es decir, de decidir cuál proceso debe ejecutarse cuándo y durante cuánto tiempo, es muy importante; lo examinaremos más adelante en este capítulo. Se han inventado muchos algoritmos para tratar de equilibrar las exigencias opuestas de eficiencia del sistema global y de equitatividad para los procesos individuales.

La transición 4 ocurre cuando acontece el suceso externo que un proceso estaba esperando (como la llegada de entradas). Si ningún otro proceso se está ejecutando en ese instante, se dispara de inmediato la transición 3, y el proceso comienza a ejecutarse. En caso contrario, el proceso tal vez tenga que esperar en el estado listo durante cierto tiempo hasta que la CPU esté disponible.

Usando el modelo de procesos, es mucho más fácil visualizar lo que está sucediendo dentro del sistema. Algunos de los procesos ejecutan programas que llevan a cabo comandos tecleados por un usuario. Otros procesos forman parte del sistema y se encargan de tareas tales como atender solicitudes de servicios de archivos o manejar los detalles de la operación de una unidad de disco o de cinta.

Cuando ocurre una interrupción de disco, el sistema toma la decisión de dejar de ejecutar el proceso en curso y ejecutar el proceso de disco, que estaba bloqueado esperando dicha interrupción. Así, en lugar de pensar en interrupciones, podemos pensar en procesos de usuario, procesos de disco, procesos de terminal, etc., que se bloquean cuando están esperando que algo suceda. Cuando se ha leído el bloque de disco o se ha tecleado el carácter, el proceso que lo estaba esperando se desbloquea y es elegible para ejecutarse otra vez.

Esta perspectiva da lugar al modelo que se muestra en la Fig. 2-3. Aquí, el nivel más bajo de sistema operativo es el planificador, con diversos procesos arriba de él. Todo el manejo de las interrupciones y los detalles del inicio y la detención de procesos están ocultos en el planificador, que en realidad es muy pequeño. El resto del sistema operativo está estructurado en forma de procesos. El modelo de la Fig. 2-3 se emplea en MINIX, con el entendido de que “planificador” no sólo se refiere a planificación de procesos, sino también a manejo de interrupciones y toda la comunicación entre procesos. No obstante, como una primera aproximación, sí muestra la estructura básica.

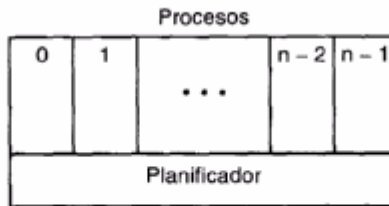


Figura 2-3. La capa más baja de un sistema operativo con estructura de procesos maneja las interrupciones y la planificación. Por encima de esa capa están los procesos secuenciales.

- a. Dentro del disco y los procesos del sistema operativo se pueden manejar alternamente un espacio para realizar intercambio con la memoria principal, lo que da origen a dos estados adicionales en el disco llamados: listo suspendido y bloqueado suspendido. Lo anterior amplifica el modelo de tres estados a un modelo de 5 que incluye el intercambio entre la memoria y el disco

◆ El Modelo De 5 Estados

- ◆ Si el Sistema Operativo se queda sin recursos el proceso se queda en 'Nuevo'.
- ◆ Cuando recibe suficientes recursos pasa al estado 'Listo'.
- ◆ Este ya es un proceso completamente funcional, aunque el Sistema Operativo no permite que haya más de un proceso ejecutándose simultáneamente.
- ◆ Cuando el Sistema Operativo permite continuar el proceso, este pasa al estado de 'Ejecución'.
- ◆ Una vez finalizada la ejecución, el proceso acaba en el estado 'Terminado'.
- ◆ Durante ciertos instantes se mantiene la información del proceso en el PCB, con el objetivo de obtener datos sobre este.
- ◆ Por último el Sistema Operativo repite todos los pasos anteriores con el resto de procesos, que se encontraban 'en espera'.
- ◆ Un estado especial sería 'bloqueado', por el cual el proceso es interrumpido hasta que cumpla ciertos requisitos.

◆ Implementación de procesos

Para implementar el modelo de procesos, el sistema operativo mantiene una tabla (un arreglo de estructuras) llamada tabla de procesos, con una entrada por cada proceso. Esta entrada contiene información acerca del estado del proceso, su contador de programa, el apuntador de pila, el reparto de memoria, la situación de sus archivos abiertos, su información de contabilidad y planificación y todos los demás aspectos de un proceso que se deben guardar cuando éste se conmuta del estado ejecutándose al estado listo, a fin de poder reiniciarlo después como si nunca se hubiera detenido.

En MINIX la administración de procesos, de memoria y de archivos corren a cargo de módulos individuales dentro del sistema, por lo que la tabla de procesos se divide en particiones y cada módulo mantiene los campos que necesita. En la Fig. 2-4 se muestran algunos de los campos más importantes. Los campos de la primera columna son los únicos pertinentes para este capítulo. Las otras dos columnas se incluyen sólo para dar una idea de qué información se necesita en otras partes del sistema.

Ahora que hemos examinado la tabla de procesos, podemos explicar un poco más la forma de cómo se mantiene la ilusión de múltiples procesos secuenciales en una máquina con una CPU y muchos dispositivos de E/S. Lo que sigue es técnicamente una descripción de cómo funciona el “planificador” de la Fig. 2-3 en MINIX, pero la mayor parte de los sistemas operativos modernos funcionan básicamente de la misma manera. Cada clase de dispositivo de E/S (p. ej., discos flexibles, discos duros, cronómetros, terminales) tiene asociada una posición cerca de la base de la memoria llamada vector de interrupción que contiene la dirección del procedimiento de servicio de interrupciones.

Supongamos que el proceso de usuario 3 se está ejecutando cuando ocurre una interrupción de disco. El hardware de interrupciones mete el contador de programa, la palabra de estado del programa y quizá uno o más registros en la pila (actual). A continuación, la computadora salta a la dirección especificada en el vector de interrupciones de disco. Esto es todo lo que el hardware hace. De aquí en adelante, el software decide lo que se hace.

Lo primero que hace el procedimiento de servicio de interrupciones es guardar todos los registros de la entrada de tabla de procesos para el proceso actual. El número del proceso actual y un apuntador a su entrada de la tabla se guardan en variables globales a fin de poder encontrarlos rápidamente. Luego se saca de la pila la información depositada en ella por la interrupción, y se ajusta el apuntador de la pila de modo que apunte a una pila temporal empleada por el manejador de procesos. Las acciones como guardar los registros y ajustar el apuntador de la pila ni siquiera pueden expresarse en C, así que son realizadas por una pequeña rutina escrita en lenguaje de

ensamblador. Una vez que esta rutina termina, invoca a un procedimiento en C para que se encargue del resto del trabajo.

| Administración de procesos | Administración de memoria | Administración de archivos |
|-----------------------------------|--------------------------------|-----------------------------------|
| Registros | Apuntador al segmento de texto | Máscara UMASK |
| Contador de programa | Apuntador al segmento de datos | Directorio raíz |
| Palabra de estado del programa | Apuntador al segmento bss | Directorio de trabajo |
| Apuntador a la pila | Estado de salida | Descriptores de archivos |
| Estado del proceso | Estado de señales | Uid efectivo |
| Hora en que se inició el proceso | Identificador del proceso | Gid efectivo |
| Tiempo de CPU usado | Proceso padre | Parámetros de llamadas al sistema |
| Tiempo de CPU de los hijos | Grupo de procesos | Diversos bits de bandera |
| Hora de la siguiente alarma | Uid real | |
| Apuntadores a la cola de mensajes | Uid efectivo | |
| Bits de señales pendientes | Gid real | |
| Identificador del proceso | Gid efectivo | |
| Diversos bits de bandera | Mapas de bits para señales | |
| | Diversos bits de bandera | |

Figura 2-4. Algunos de los campos de la tabla de procesos de MINIX.

La comunicación entre procesos en MINIX se efectúa a través de mensajes, así que el siguiente paso consiste en construir un mensaje para enviarlo al proceso de disco, el cual estará bloqueado en espera de recibirlo. El mensaje dice que ocurrió una interrupción, a fin de distinguirlo de los mensajes de usuarios que solicitan la lectura de bloques de disco y cosas por el estilo. Ahora se cambia el estado del proceso de disco de bloqueado a listo y se llama al planificador. En MINIX, los diferentes procesos tienen diferentes prioridades, con objeto de dar mejor servicio a los manejadores de dispositivos de E/S que a los procesos de usuario. Si el proceso de disco es ahora el proceso ejecutable con la prioridad más alta, se planificará para ejecutarse. Si el proceso que se interrumpió tiene la misma importancia o una mayor, se le planificará para ejecutarse otra vez, y el proceso de disco tendrá que esperar un poco.

En cualquier caso, ahora regresa el procedimiento en C invocado por el código de interrupción en lenguaje de ensamblador, y este código carga los registros y el mapa de memoria para el proceso que ahora es el actual, y lo pone en marcha. El manejo de interrupciones y la planificación se resumen en la Fig. 2-5. Vale la pena señalar que los detalles varían un poco de un sistema a otro.

1. El hardware agrega a la pila el contador de programa, etc.
2. El hardware carga un nuevo contador de programa del vector de interrupciones.
3. El procedimiento en lenguaje ensamblador guarda los registros.
4. El procedimiento en lenguaje ensamblador prepara una nueva pila.
5. Se ejecuta el servicio de interrupciones en C (por lo regular lee y coloca en buffers las entradas).
6. El planificador marca la tarea en espera como lista.
7. El planificador decide cuál proceso debe ejecutarse a continuación.
8. El procedimiento en C regresa al código en ensamblador.
9. El procedimiento en lenguaje ensamblador inicia el nuevo proceso actual.

Figura 2-5. Bosquejo de lo que el nivel más bajo del sistema operativo hace cuando ocurre una interrupción.

◆ Hilos

En un proceso tradicional del tipo que acabamos de estudiar hay un solo hilo de control y un solo contador de programa en cada proceso. Sin embargo, algunos sistemas operativos modernos manejan múltiples hilos de control dentro de un proceso. Estos hilos de control normalmente se llaman sólo hilos u, ocasionalmente, procesos ligeros. En la Fig. 2-6(a) vemos tres procesos tradicionales. Cada proceso tiene su propio espacio de direcciones y un solo hilo de control. En contraste, en la Fig. 2-6(b) vemos un solo proceso con tres hilos

De control. Aunque en ambos casos tenemos tres hilos, en la Fig. 2-6(a) cada uno de ellos opera en un espacio de direcciones distinto, en tanto que en la Fig. 2-6(b) los tres comparten el mismo espacio de direcciones.

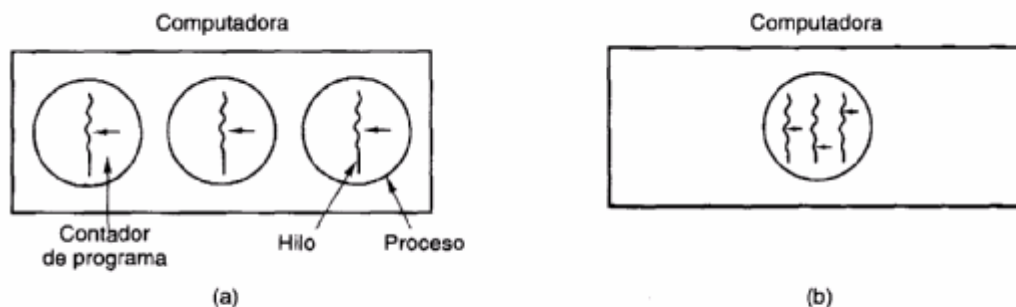


Figura 2-6. (a) Tres procesos, cada uno con un hilo. (b) Un proceso con tres hilos.

Como ejemplo de situación en la que podrían usarse múltiples hilos, consideremos un proceso servidor de archivos que recibe solicitudes para leer y escribir archivos y devuelve los datos solicitados o acepta datos actualizados. A fin de mejorar el rendimiento, el servidor mantiene un caché de archivos recientemente usados en la memoria, leyendo del caché y escribiendo en él siempre que es posible. Esta situación se presta bien al modelo de la Fig. 2.6 (b). Cuando llega una solicitud, se le entrega a un hilo para que la procese. Si ese hilo se bloquea en algún momento esperando una Esta situación se presta bien al modelo de la Fig. 2.6 (b). Cuando llega una solicitud, se le entrega a un hilo para que la procese. Si ese hilo se bloquea en algún momento esperando

una transferencia de disco, los demás hilos aún pueden ejecutarse, de modo que el servidor puede seguir procesando nuevas solicitudes incluso mientras se está efectuando E/S de disco. En cambio, el modelo de la Fig. 2-6(a) no es apropiado, porque es indispensable que todos los hilos del servidor de archivos tengan acceso al mismo caché, y los tres hilos de la Fig. 2-6(a) no comparten el mismo espacio de direcciones y, por tanto, no pueden compartir el mismo caché en memoria.

Otro ejemplo de caso en el que son útiles los hilos es el de los navegadores de la World Wide Web, como Netscape y Mosaic. Muchas páginas Web contienen múltiples imágenes pequeñas. Para cada imagen de una página Web, el navegador debe establecer una conexión individual con el sitio de la página de casa y solicitar la imagen. Se desperdicia una gran cantidad de tiempo estableciendo y liberando todas estas conexiones. Si tenemos múltiples hilos dentro del navegador, podemos solicitar muchas imágenes al mismo tiempo, acelerando considerablemente el rendimiento en la mayor parte de los casos, ya que en el caso de imágenes pequeñas el tiempo de preparación es el factor limitante, no la rapidez de la línea de transmisión.

Cuando están presentes múltiples hilos en el mismo espacio de direcciones, algunos de los campos de la Fig. 2-4 no contienen información para cada proceso, sino para cada hilo, así que se necesita una tabla de hilos aparte, con una entrada por hilo. Entre los elementos que son distintos para cada hilo están el contador de programa, los registros y el estado. El contador de programa se necesita porque los hilos, al igual que los procesos, pueden suspenderse y reanudarse. Los registros se necesitan porque cuando los hilos se suspenden sus registros deben guardarse. Por último, los hilos, al igual que los procesos, pueden estar en los estados de ejecutándose, listo o bloqueado.

En algunos sistemas el sistema operativo no está consciente de la existencia de los hilos. En otras palabras, los hilos se manejan totalmente en el espacio de usuario. Por ejemplo, cuando un hilo está a punto de bloquearse, escoge e inicia a su sucesor antes de detenerse. Hay varios paquetes de hilos a nivel de usuario, incluidos los paquetes de hilos P de POSIX e hilos C de Mach.

En otros sistemas, el sistema operativo está consciente de la existencia de múltiples hilos por proceso, así que, cuando un hilo se bloquea, el sistema operativo escoge el que se ejecutará a continuación, ya sea del mismo proceso o de uno distinto. Para realizar la planificación, el kernel debe tener una tabla de hilos que liste todos los hilos del sistema, análoga a la tabla de procesos.

Aunque estas dos alternativas pueden parecer equivalentes, su rendimiento es muy distinto. La conmutación de hilos es mucho más rápida cuando la administración de hilos se efectúa en el espacio de usuario que cuando se necesita una llamada al kernel. Este hecho es un argumento convincente para realizar la administración de hilos en el espacio de usuario. Por otro lado, cuando los hilos se manejan totalmente en el espacio de usuario y uno se bloquea (p. ej., esperando E/S o que se maneje una falla de página), el kernel bloquea todo el proceso, ya que no tiene idea de que existen otros hilos. Este hecho es un argumento importante en favor de realizar la administración

de hilos en el kernel. La consecuencia es que se usan ambos sistemas; además, se han propuesto diversos esquemas híbridos (Anderson et al., 1992).

Sea que los hilos sean administrados por el kernel o en el espacio de usuario, introducen una serie de problemas que deben resolverse y que modifican sustancialmente el modelo de programación. Para comenzar, consideremos los efectos de la llamada al sistema FORK. Si el proceso padre tiene múltiples hilos, ¿debe tenerlos también el hijo? Si no, es posible que el proceso no funcione correctamente, ya que es posible que todos ellos sean esenciales.

Por otro lado, si el proceso hijo obtiene tantos hilos como el padre, ¿qué sucede si un hilo estaba bloqueado en una llamada READ de, digamos, el teclado. ¿Hay ahora dos hilos bloqueados esperando el teclado? Si se teclea una línea, ¿reciben ambos hilos una copia de ella? ¿Sólo el padre? ¿Sólo el hijo? El mismo problema ocurre con las conexiones de red abiertas.

Otra clase de problemas tiene que ver con el hecho de que los hilos comparten muchas estructuras de datos. ¿Qué sucede si un hilo cierra un archivo mientras otro todavía lo está leyendo? Supongamos que un hilo se da cuenta de que hay muy poca memoria y comienza a asignar más memoria. Luego, antes de que termine de hacerlo, ocurre una conmutación de hilo, y el nuevo hilo también se da cuenta de que hay demasiada poca memoria y comienza a asignar más memoria. ¿La asignación ocurre una sola vez o dos? En casi todos los sistemas que no se diseñaron pensando en hilos, las bibliotecas (como el procedimiento de asignación de memoria) no son reentrantes, y se caen si se emite una segunda llamada mientras la primera todavía está activa.

Otro problema se relaciona con los informes de error. En UNIX, después de una llamada al sistema, la situación de la llamada se coloca en una variable global, errno. ¿Qué sucede si un hilo efectúa una llamada al sistema y, antes de que pueda leer errno, otro hilo realiza una llamada al sistema, borrando el valor original? Consideremos ahora las señales. Algunas señales son lógicamente específicas para cada hilo, en tanto que otras no lo son. Por ejemplo, si un hilo invoca ALARM, tiene sentido que la señal resultante se envíe al hilo que efectuó la llamada. Si el kernel está consciente de la existencia de hilos, normalmente puede asegurarse de que el hilo correcto reciba la señal. Si el kernel no sabe de los hilos, el paquete de hilos de alguna forma debe seguir la pista a las alarmas. Surge una complicación adicional en el caso de hilos a nivel de usuario cuando (como sucede en UNIX) un proceso sólo puede tener una alarma pendiente a la vez y varios hilos pueden invocar ALARM independientemente.

Otras señales, como una interrupción de teclado, no son específicas para un hilo. ¿Quién deberá atraparlas? ¿Un hilo designado? ¿Todos los hilos? ¿Un hilo recién creado? Todas estas soluciones tienen problemas. Además, ¿qué sucede si un hilo cambia los manejadores de señales sin avisar a los demás hilos’?

Un último problema introducido por los hilos es la administración de la pila. En muchos sistemas, cuando hay un desbordamiento de pila, el kernel simplemente amplía la pila automáticamente. Cuando un proceso tiene varios hilos, también debe tener varias pilas. Si el kernel no tiene conocimiento de todas estas pilas, no podrá hacerlas crecer automáticamente cuando haya una falla de pila. De hecho, es posible que el kernel ni siquiera se dé cuenta de que la falla de memoria está relacionada con el crecimiento de una pila.

Estos problemas ciertamente no son insuperables, pero sí ponen de manifiesto que la simple introducción de hilos en un sistema existente sin un rediseño sustancial del sistema no funcionará. Cuando menos, es preciso redefinir la semántica de las llamadas al sistema y reescribir las bibliotecas. Además, todas estas cosas deben hacerse de modo tal que sigan siendo compatibles hacia atrás con los programas existentes para el caso limitante de un proceso con un solo hilo. Si desea información adicional acerca de los hilos, véase (Hauser et al., 1993; y Marsh et al., 1991).

<http://laurel.datsi.fi.upm.es/docencia/asignaturas/sod>

Andrew S. Tanenbaum: "Sistemas Operativos Distribuidos", Prentice Hall (1996).

3.1.1. Comunicación Entre Procesos

Los procesos con frecuencia necesitan comunicarse con otros procesos. Por ejemplo, en un conducto de shell, la salida del primer proceso debe pasarse al segundo proceso, y así sucesivamente. Por tanto, es necesaria la comunicación entre procesos, de preferencia en una forma bien estructurada que no utilice interrupciones. En las siguientes secciones examinaremos algunos de los problemas relacionados con esta comunicación entre procesos o IPC.

En pocas palabras, tenemos tres problemas. Ya hicimos alusión al primero de ellos: ¿cómo puede un proceso pasar información a otro? El segundo tiene que ver con asegurarse de que dos o más procesos no se estorben mutuamente al efectuar actividades críticas (suponga que dos procesos tratan de apoderarse al mismo tiempo de los últimos 100K de memoria). El tercero se relaciona con la secuencia correcta cuando existen dependencias: Si el proceso A produce datos y el proceso B los imprime, B tiene que esperar hasta que A haya producido algunos datos antes de comenzar a imprimir. Examinaremos estos tres problemas a partir de la siguiente sección.

3.1.1.1 Condiciones de competencia

En algunos sistemas operativos, los procesos que están colaborando podrían compartir cierto almacenamiento común en el que ambos pueden leer y escribir. El almacenamiento compartido puede estar en la memoria principal o puede ser un archivo compartido; la ubicación de la memoria compartida no altera la naturaleza de la comunicación ni los problemas que surgen. Para ver cómo funciona la comunicación entre procesos en la práctica, consideremos un ejemplo

sencillo pero común, un spooler de impresión. Cuando un proceso desea imprimir un archivo, introduce el nombre del archivo en un directorio de spooler especial. Otro proceso, el demonio de impresión, revisa periódicamente el directorio para ver si hay archivos por imprimir, y si los hay los imprime y luego borra sus nombres del directorio.

Imagine que nuestro directorio de spooler tiene un número elevado (potencialmente infinito) de ranuras, numeradas 0, 1, 2,..., cada una con capacidad para un nombre de archivo. Imagine además que hay dos variables compartidas, *out*, que apuntan al siguiente archivo por imprimir, e *in*, que apunta a la siguiente ranura libre del directorio. Estas dos variables bien podrían guardarse en un archivo de dos palabras que estuviera accesible para todos los procesos. En un instante dado, las ranuras 0 a 3 están vacías (los archivos ya se imprimieron) y las ranuras 4 a 6 están llenas (con los nombres de archivos en cola para imprimirse). En forma más o menos simultánea, los procesos A y B deciden que desean poner en cola un archivo para impresión. Esta situación se muestra en la Fig. 2-7.

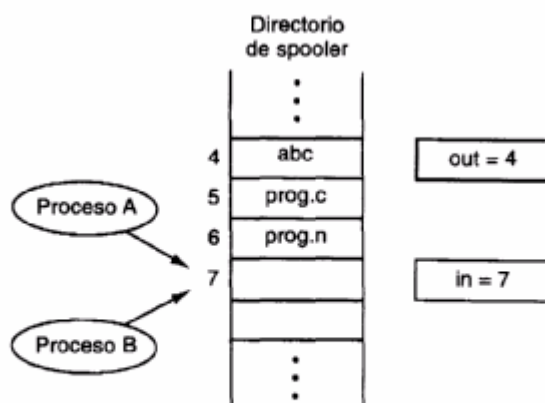


Figura 2-7. Dos procesos quieren acceder a memoria compartida al mismo tiempo.

En las jurisdicciones en las que aplica la ley de Murphy, podría ocurrir lo siguiente. El proceso A lee *in* y almacena su valor, 7, en una variable local llamada *siguiente_ranura_libre*. Justo en ese momento ocurre una interrupción de reloj y la CPU decide que el proceso A ya se ejecutó durante suficiente tiempo, así que conmuta al proceso B. El proceso B también lee *in*, y también obtiene un 7, así que almacena el nombre de su archivo en la ranura 7 y actualiza *in* con el valor 8. Luego se va y hace otras cosas.

Tarde o temprano, el proceso A se ejecuta otra vez, continuando en donde se interrumpió. A examina *siguiente_ranura_libre*, encuentra 7 ahí, y escribe su nombre de archivo en la ranura 7, borrando el nombre que el proceso B acaba de poner ahí. Luego A calcula *siguiente_ranura_libre* + 1, que es 8, y asigna 8 a *in*. El directorio de spooler no tiene contradicciones internas, así que el demonio de impresión no notará que algo esté mal, si bien el proceso B nunca obtendrá sus salidas. Situaciones como ésta, en la que dos o más procesos leen o escriben datos compartidos y el resultado final depende de quién se ejecuta precisamente cuándo, se denominan condiciones

de competencia. La depuración de programas que contienen condiciones de competencia no es nada divertida. Los resultados de la mayor parte de las pruebas son correctos, pero de vez en cuando algo raro e inexplicable sucede.

◆ Secciones críticas

¿Cómo evitamos las condiciones de competencia? La clave para evitar problemas en ésta y muchas otras situaciones en las que se comparte memoria, archivos o cualquier otra cosa es encontrar una forma de prohibir que más de un proceso lea y escriba los datos compartidos al mismo tiempo. Dicho de otro modo, lo que necesitamos es exclusión mutua: alguna forma de asegurar que si un proceso está usando una variable o archivo compartido, los otros procesos quedarán excluidos de hacer lo mismo.

El problema anterior ocurrió porque el proceso B comenzó a usar una de las variables compartidas antes de que A terminara de usarla. La selección de operaciones primitivas apropiadas para lograr la exclusión mutua es un aspecto importante del diseño de cualquier sistema operativo, y un tema que examinaremos con gran detalle en las siguientes secciones.

El problema de evitar condiciones de competencia también puede formularse de manera abstracta. Una parte del tiempo, un proceso está ocupado realizando cálculos internos y otras cosas que no dan pie a condiciones de competencia. Sin embargo, hay veces en que un proceso está accediendo a memoria o archivos compartidos, o efectuando otras tareas críticas que pueden dar lugar a competencias. Esa parte del programa en la que se accede a la memoria compartida se denomina región crítica o sección crítica. Si pudiéramos organizar las cosas de modo que dos procesos nunca pudieran estar en sus regiones críticas al mismo tiempo, podríamos evitar las condiciones de competencia.

Aunque este requisito evita las condiciones de competencia, no es suficiente para lograr que los procesos paralelos cooperen de manera correcta y eficiente usando datos compartidos. Necesitamos que se cumplan cuatro condiciones para tener una buena solución:

1. Dos procesos nunca pueden estar simultáneamente dentro de sus regiones críticas.
2. No puede suponerse nada acerca de las velocidades o el número de las CPU.
3. Ningún proceso que se ejecute fuera de su región crítica puede bloquear a otros procesos.
4. Ningún proceso deberá tener que esperar indefinidamente para entrar en su región crítica.

◆ **Exclusión mutua con espera activa**

En esta sección examinaremos varias propuestas para lograr la exclusión mutua, de modo que cuando un proceso esté ocupado actualizando la memoria compartida en su región crítica, ningún otro proceso entre en su región crítica y cause problemas.

◆ **Inhabilitación de interrupciones**

La solución más sencilla es hacer que cada proceso inhabilite las interrupciones justo después de ingresar en su región crítica y vuelva a habilitarlas justo antes de salir de ella. Con las interrupciones inhabilitadas, no pueden ocurrir interrupciones de reloj. Después de todo, la CPU sólo se conmuta de un proceso a otro como resultado de interrupciones de reloj o de otro tipo, y con las interrupciones desactivadas la CPU no se conmutará a ningún otro proceso. Así, una vez que un proceso ha inhabilitado las interrupciones, puede examinar y actualizar la memoria compartida sin temor a que otro proceso intervenga.

Este enfoque casi nunca resulta atractivo porque no es prudente conferir a los procesos de usuario la facultad de desactivar las interrupciones. Supongamos que uno de ellos lo hiciera, y nunca habilitara las interrupciones otra vez. Esto podría terminar con el sistema. Además, si el sistema es multiprocesador, con dos o más CPU, la inhabilitación de las interrupciones afectaría sólo a la CPU que ejecutara la instrucción de inhabilitación; las demás seguirían ejecutándose y podrían acceder a la memoria compartida.

Por otro lado, en muchos casos es necesario que el kernel mismo inhabilite las interrupciones durante unas cuantas instrucciones mientras actualiza variables o listas. Si ocurriera una interrupción en un momento en que la lista de procesos listos, por ejemplo, está en un estado inconsistente, ocurrirían condiciones de competencia. La conclusión es: la inhabilitación de interrupciones suele ser una técnica útil dentro del sistema operativo mismo pero no es apropiada como mecanismo de exclusión mutua general para los procesos de usuario.

◆ **Variables de candado**

Veamos ahora una posible solución de software. Supongamos que tenemos una sola variable (de candado) compartida cuyo valor inicial es 0. Cuando un proceso quiere entrar en su región crítica, lo primero que hace es probar el candado. Si el candado es 0, el proceso le asigna 1 y entra en su región crítica; si es 1, el proceso espera hasta que el candado vuelve a ser 0. Así, un 0 significa que ningún proceso está en su región crítica, y un 1 significa que algún proceso está en su región crítica.

Desafortunadamente, esta idea contiene exactamente el mismo defecto fatal que vimos en el directorio de spooler. Supongamos que un proceso lee el candado y ve que es 0. Antes de que este proceso pueda asignar 1 al candado, se planifica otro proceso, el cual se ejecuta y asigna 1 al candado. Cuando el primer proceso continúa su ejecución, asignará 1 al candado, y dos procesos estarán en su región crítica al mismo tiempo.

Podríamos pensar que este problema puede superarse leyendo primero el valor del candado, y verificándolo otra vez justo antes de guardar el 1 en él, pero esto no sirve de nada. La competencia ocurriría entonces si el segundo proceso modifica el candado justo después de que el primer proceso terminó su segunda verificación.

◆ Alternancia estricta

Una tercera estrategia para abordar el problema de la exclusión mutua se muestra en la Fig. 2-8. Este fragmento de programa, como casi todos los del libro, está escrito en C. Se escogió C aquí porque los sistemas operativos reales con frecuencia se escriben en C (u ocasionalmente en C++), pero casi nunca en lenguajes como Modula 2 o Pascal.

```
while (TRUE) {  
    while(turn != 0) /* esperar */ ;  
    critical_region();  
    turn = 1;  
    noncritical_region();  
}
```

(a)

```
while (TRUE) {  
    while(turn != 1) /* esperar */ ;  
    critical_region();  
    turn = 0;  
    noncritical_region();  
}
```

(b)

Figura 2-8. Solución propuesta para el problema de la región crítica.

En la Fig. 2-8, la variable interna `turn`, que inicialmente es 0, indica a quién le toca entrar en la región crítica y examinar o actualizar la memoria compartida. En un principio, el proceso 0 inspecciona `turn`, ve que es 0, y entra en su región crítica. El proceso 1 también ve que `turn` es 0 y se mantiene en un ciclo corto probando `turn` continuamente para detectar el momento en que cambia a 1. Esta prueba continua de una variable hasta que adquiere algún valor se denomina espera activa, y normalmente debe evitarse, ya que desperdicia tiempo de CPU. La espera activa sólo debe usarse cuando exista una expectativa razonable de que la espera será corta.

Cuando el proceso 0 sale de la región crítica, asigna 1 a `turn`, a fin de que el proceso 1 pueda entrar en su región crítica. Supongamos que el proceso 1 termina su región crítica rápidamente, de modo que ambos procesos están en sus regiones no críticas, y `turn` vale 0. Ahora el proceso 0 ejecuta su ciclo completo rápidamente, regresando a su región no crítica después de haber

asignado 1 a turn. Luego, el proceso O termina su región no crítica y regresa al principio de su ciclo. Desafortunadamente, no puede entrar en su región crítica porque turn es 1 y el proceso 1 está ocupado en su región no crítica. Dicho de otro modo, la alternancia de turnos no es una buena idea cuando un proceso es mucho más lento que el otro.

Esta situación viola la condición 3 antes mencionada: el proceso O está siendo bloqueado por un proceso que no está en su región crítica. Volviendo al ejemplo del directorio de spooler, si ahora asociamos la región crítica a las actividades de leer y escribir el directorio de spooler, el proceso O no podría imprimir otro archivo porque el proceso 1 está haciendo alguna otra cosa.

De hecho, esta solución requiere que los dos procesos se alternen estrictamente en el ingreso a sus regiones críticas, por ejemplo, al poner archivos en spool. Ningún proceso podría poner en spool dos archivos seguidos. Si bien este algoritmo evita todas las competencias, no es en realidad un candidato serio para ser una solución porque viola la condición 3.

◆ Solución de Peterson

Combinando la idea de tomar turnos con la de tener variables de candado y variables de advertencia, un matemático holandés, T. Dekker, inventó una solución de software para el problema de la exclusión mutua que no requiere una alternancia estricta. Si desea una explicación del algoritmo de Dekker, véase (Dijkstra, 1965).

En 1981, G. L. Peterson descubrió una forma mucho más sencilla de lograr la exclusión mutua, haciendo obsoleta la solución de Dekker. El algoritmo de Peterson se muestra en la Fig. 2-9, y consiste en dos procedimientos escritos en ANSI C, lo que implica que se deben proporcionar prototipos de función para todas las funciones que se definen y usan. Sin embargo, a fin de ahorrar espacio, no mostraremos los prototipos en este ejemplo ni en los que siguen.

Antes de usar las variables compartidas (es decir, antes de entrar en su región crítica), cada proceso invoca `enter_region` (entrar en región) con su propio número de proceso, 0 o 1, como parámetro. Esta invocación lo obligará a esperar, si es necesario, hasta que pueda entrar sin peligro. Después de haber terminado de manipular las variables compartidas, el proceso invoca `leave_region` (salir de región) para indicar que ya terminó y permitir que el otro proceso entre si lo desea.

Veamos cómo funciona esta solución. En un principio ninguno de los procesos está en su región crítica. Ahora el proceso O invoca `enter_region`, e indica su interés asignando TRUE a su elemento del arreglo `interested` (interesado) y asignando turn a O. Puesto que el proceso 1 no está interesado, `enter_region` regresa de inmediato. Si ahora el proceso 1 invoca `enter_region`,

permanecerá dando vueltas en su ciclo hasta que se asigne FALSE a interested[0], cosa que sólo sucede cuando el proceso 0 invoca leave_region para salir de su región crítica.

```
#define FALSE    0
#define TRUE     1
#define N        2          /* número de procesos */

int turn;                  /* ¿a quién le toca? */
int interested[N];         /* todos los valores son inicialmente 0 (FALSE) */

void enter_region(int process); /* proceso 0 o 1 */
{
    int other;              /* número del otro proceso */

    other = 1 - process;    /* lo opuesto de process */
    interested[process] = TRUE; /* mostrar interés */
    turn = process;         /* establecer bandera */
    while (turn == process && interested[other] == TRUE) /* instrucción nula */ ;
}

void leave_region(int process) /* process: quién sale */
{
    interested[process] = FALSE; /* indicar salida de la región crítica */
}
```

Figura 2-9. Solución de Peterson para lograr la exclusión mutua.

Consideremos ahora el caso en que ambos procesos invocan enter_region casi simultáneamente. Ambos almacenarán su número de proceso en turn, pero el único que cuenta es el que se almacena después; el primero se pierde. Supongamos que el proceso 1 es el segundo en almacenar su número de proceso, así que turn vale 1. Cuando ambos procesos llegan a la instrucción while, el proceso 0 lo ejecuta cero veces e ingresa en su región crítica. El proceso 1 da vueltas en el ciclo y no entra en su región crítica.

3.1.2. Dormir y despertar

Tanto la solución de Peterson como la que usa TSL son correctas, pero ambas tienen el defecto de requerir espera activa. En esencia, lo que estas soluciones hacen es lo siguiente: cuando un proceso desea entrar en su región crítica verifica si está permitida la entrada; si no, el proceso simplemente repite un ciclo corto esperando hasta que lo esté.

Este enfoque no sólo desperdicia tiempo de CPU, sino que también puede tener efectos inesperados. Consideremos una computadora con dos procesos, H, de alta prioridad, y L, de baja prioridad. Las reglas de planificación son tales que H se ejecuta siempre que está en el estado listo.

En un momento dado, con L en su región crítica, H queda listo para ejecutarse (p. ej., se completa una operación de E/S). H inicia ahora la espera activa, pero dado que L nunca se planifica mientras H se está ejecutando, L nunca tiene oportunidad de salir de su región crítica, y H permanece en un ciclo infinito. Esta situación se conoce como problema de inversión de prioridad.

Examinemos ahora algunas primitivas de comunicación entre procesos que se bloquean en lugar de desperdiciar tiempo de CPU cuando no se les permite entrar en sus regiones críticas. Una de las más sencillas es el par SLEEP y WAKEUP. SLEEP (dormir) es una llamada al sistema que hace que el invocador se bloquee, es decir, se suspenda hasta que otro proceso lo despierte. La llamada WAKEUP (despertar) tiene un parámetro, el proceso que se debe despertar. Como alternativa, tanto SLEEP como WAKEUP pueden tener un parámetro cada uno, una dirección de memoria que sirve para enlazar los SLEEP con los WAKEUP.

◆ El problema de productor-consumidor

Como ejemplo de uso de estas primitivas, consideremos el problema de productor-consumidor (también conocido como problema del buffer limitado). Dos procesos comparten un mismo buffer de tamaño fijo. Uno de ellos, el productor, coloca información en el buffer, y el otro, el consumidor, la saca. (También es posible generalizar el problema a m productores y n consumidores, pero sólo consideraremos el caso de un productor y un consumidor porque esto simplifica las soluciones.)

Surgen problemas cuando el productor quiere colocar un nuevo elemento en el buffer, pero éste ya está lleno. La solución es que el productor se duerma y sea despertado cuando el consumidor haya retirado uno o más elementos. De forma similar, si el consumidor desea sacar un elemento del buffer y ve que está vacío, se duerme hasta que el productor pone algo en el buffer y lo despierta.

Este enfoque parece muy sencillo, pero da lugar a los mismos tipos de condiciones de competencia que vimos antes con el directorio de spooler. Para seguir la pista al número de elementos contenidos en el buffer, necesitaremos una variable, count. Si el número máximo de elementos que el buffer puede contener es N, el código del productor primero verificará si count es igual a N. Si es así, el productor se dormirá; si no, el productor agregará un elemento e incrementará count.

El código del consumidor es similar: primero se prueba count para ver si es 0. Si así es, el consumidor se duerme; si no, el consumidor saca un elemento y decrementa el contador. Cada uno de estos procesos verifica también si el otro debería estar durmiendo, y si no es así, lo despierta. El código del productor y del consumidor se muestra en la Fig. 2-11. Para expresar las llamadas al sistema como SLEEP y WAKEUP en C, las mostraremos como llamadas a rutinas de

biblioteca. Éstas no forman parte de la biblioteca estándar de C, pero es de suponer que estarían disponibles en cualquier sistema que realmente tuviera esas llamadas al sistema. Los procedimientos `enter_item` (colocar elemento) y `remove_item` (retirar elemento), que no se muestran, se encargan de la contabilización de la colocación de elementos en el buffer y el retiro de elementos de él.

```
#define N 100                                /* número de ranuras del buffer */
int count = 0;                               /* número de elementos en el buffer */

void producer(void)
{
    while (TRUE) {                           /* repetir indefinidamente */
        produce_item();                     /* generar el siguiente elemento */
        if (count == N) sleep();            /* si el buffer está lleno, dormir */
        enter_item();                       /* colocar elemento en el buffer */
        count = count + 1;                  /* incrementar la cuenta de elementos
                                           en el buffer */

        if (count == 1) wakeup(consumer);  /* ¿estaba vacío el buffer? */
    }
}

void consumer(void)
{
    while (TRUE){                           /* repetir indefinidamente */
        if (count == 0) sleep();            /* si el buffer está vacío, dormir */
        remove_item();                     /* remover elemento del buffer */
        count = count -1;                  /* decrementar la cuenta de elementos
                                           en el buffer */

        if (count == N-1) wakeup(producer); /* ¿estaba lleno el buffer? */
        consume_item();                    /* imprimir elemento */
    }
}
```

Figura 2-11. El problema de productor-consumidor con una condición de competencia fatal.

Volvamos ahora a la condición de competencia. Ésta puede ocurrir porque el acceso a `count` es inestricto, y podría presentarse la siguiente situación. El buffer está vacío y el consumidor acaba de leer `count` para ver si es 0. En ese instante, el planificador decide dejar de ejecutar el consumidor temporalmente y comenzar a ejecutar el productor. Éste coloca un elemento en el buffer, incrementa `count`, y observa que ahora vale 1. Esto implica que antes `count` valía 0, y por ende que el consumidor está durmiendo, así que el productor invoca `wakeup` para despertar al consumidor.

Desafortunadamente, el consumidor todavía no está dormido lógicamente, de modo que la señal de despertar se pierde. Cuando el consumidor reanuda su ejecución, prueba el valor de `count` que

había leído previamente, ve que es O y se duerme. Tarde o temprano el productor llenará el buffer y se dormirá. Ambos seguirán durmiendo eternamente.

La esencia del problema aquí es que se perdió una llamada enviada para despertar a un proceso que (todavía) no estaba dormido. Si no se perdiera, todo funcionaría. Una compostura rápida consiste en modificar las reglas y agregar un bit de espera de despertar a la escena. Cuando se envía una llamada de despertar a un proceso que está despierto, se enciende este bit. Después, cuando el proceso trata de dormirse, si el bit de espera de despertar está encendido, se apagará, pero el proceso seguirá despierto. El bit de espera de despertar actúa como una alcancía de señales de despertar. Aunque el bit de espera de despertar nos salva el pellejo en este ejemplo, es fácil construir ejemplos con tres o más procesos en los que un bit de espera de despertar es insuficiente. Podríamos crear otro parche y agregar un segundo bit de espera de despertar, o quizá 8 o 32, pero en principio el problema sigue ahí.

◆ Semáforos

Ésta era la situación en 1965, cuando E. W. Dijkstra (1965) sugirió usar una variable entera para contar el número de señales de despertar guardadas para uso futuro. En esta propuesta se introdujo un nuevo tipo de variable, llamada semáforo. Un semáforo podía tener el valor O, indicando que no había señales de despertar guardadas, o algún valor positivo si había una o más señales de despertar pendientes.

Dijkstra propuso tener dos operaciones, DOWN y UP (generalizaciones de SLEEP y WAKEUP, respectivamente). La operación DOWN (abajo) aplicada a un semáforo verifica si el valor es mayor que O; de ser así, decrementa el valor (esto es, gasta una señal de despertar almacenada) y continúa. Si el valor es O, el proceso se pone a dormir sin completar la operación DOWN por el momento. La verificación del valor, su modificación y la acción de dormirse, si es necesaria, se realizan como una sola acción atómica indivisible. Se garantiza que una vez que una operación de semáforo se ha iniciado, ningún otro proceso podrá acceder al semáforo hasta que la operación se haya completado o bloqueado. Esta atomicidad es absolutamente indispensable para resolver los problemas de sincronización y evitar condiciones de competencia.

La operación UP incrementa el valor del semáforo direccionado. Si uno o más procesos están durmiendo en espera de ese semáforo, imposibilitados de completar una operación DOWN previa, el sistema escoge uno de ellos (p. ej., al azar) y le permite completar su DOWN. Así, después de un up con un semáforo que tiene procesos durmiendo esperando, el semáforo seguirá siendo O, pero habrá un proceso menos que se halle en fase de durmiendo esperando. La operación de incrementar el semáforo y despertar un proceso también es indivisible. Ningún proceso se bloquea durante un up, así como ningún

Proceso se bloquea realizando un WAKEUP en el modelo anterior.

Como acotación, en su artículo original Dijkstra usó las letras P y en lugar de DOWN y UP, respectivamente, pero en vista de que éstos no tienen significado mnemónico para quienes no hablan holandés (y apenas un significado marginal para quienes lo hablan), usaremos los términos DOWN y up en vez de éstos. DOWN y UP se introdujeron por primera vez en Algol 68.

Resolución del problema de productor-consumidor usando semáforos.

Los semáforos resuelven el problema de la señal de despertar perdida, como se muestra en la Fig. 2-12. Es indispensable que se implementen de modo que sean indivisibles. El método normal consiste en implementar UP y DOWN como llamadas al sistema, para que el sistema operativo inhabilite brevemente todas las interrupciones mientras prueba el semáforo, lo actualiza y pone el proceso a dormir, si es necesario. Todas estas acciones requieren sólo unas cuantas instrucciones, así que la inhabilitación de las interrupciones no tiene consecuencias adversas. Si se están usando múltiples CPU, cada semáforo debe estar protegido con una variable de candado, usando la instrucción TSL para asegurarse de que sólo una CPU a la vez examine el semáforo. Cerciórese de entender que el empleo de TSL para evitar que varias CPU accedan al semáforo al mismo tiempo es muy diferente de la espera activa del productor o el consumidor cuando esperan que el otro vacíe o llene el buffer. La operación del semáforo sólo toma unos cuantos microsegundos, mientras que el productor o el consumidor podrían tardar un tiempo arbitrariamente largo.

```
#define N 100                                /* número de ranuras del buffer */
typedef int semaphore;                       /* los semáforos son un tipo especial de int */
semaphore mutex = 1;                         /* controla el acceso a la región crítica */
semaphore empty = N;                        /* cuenta las ranuras de buffer vacías */
semaphore full = 0;                          /* cuenta las ranuras de buffer llenas */

void producer(void)
{
    int item;

    while (TRUE) {                          /* TRUE es la constante 1 */
        produce_item(&item);                /* generar algo para ponerlo en el buffer */
        down(&empty);                       /* decrementar el contador empty */
        down(&mutex);                       /* entrar en la región crítica */
        enter_item(item);                   /* colocar el nuevo elemento en el buffer */
        up(&mutex);                         /* salir de la región crítica */
        up(&full);                          /* incrementar el contador de ranuras llenas */
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {                          /* ciclo infinito */
        down(&full);                        /* decrementar el contador full */
        down(&mutex);                       /* entrar en la región crítica */
        remove_item(&item);                 /* sacar elemento del buffer */
        up(&mutex);                         /* salir de la región crítica */
        up(&empty);                         /* incrementar el contador de ranuras vacías */
        consume_item(item);                 /* hacer algo con el elemento */
    }
}
```

Figura 2-12. El problema de productor-consumidor usando semáforos.

Esta solución usa tres semáforos: uno llamado full para contar el número de ranuras que están llenas, uno llamado empty para contar el número de ranuras que están vacías, y otro llamado mutex para asegurarse de que el productor y el consumidor no accedan al buffer al mismo tiempo. Full inicialmente vale 0, empty inicialmente es igual al número de ranuras del buffer y mutex inicialmente es 1. Los semáforos a los que se asignan 1 como valor inicial y son utilizados por dos o más procesos para asegurar que sólo uno de ellos pueda entrar en su región crítica al mismo tiempo se denominan semáforos binarios. Si cada proceso ejecuta DOWN justo antes de entrar en su región crítica, y up justo después de salir de ella, la exclusión mutua está garantizada.

Ahora que contamos con una buena primitiva de comunicación entre procesos, regresemos y examinemos otra vez la secuencia de interrupción de la Fig. 2-5. En un sistema que usa semáforos, la forma natural de ocultar las interrupciones es tener un semáforo, inicialmente puesto en 0,

asociado a cada dispositivo de E/S. Inmediatamente después de iniciar un dispositivo de E/S, el proceso que lo administra ejecuta DOWN con el semáforo correspondiente, bloqueándose así de inmediato. Cuando llega la interrupción, el manejador de instrucciones ejecuta up con el semáforo correspondiente, haciendo que el proceso en cuestión quede otra vez listo para ejecutarse.

En este modelo, el paso 6 de la Fig. 2-5 consiste en ejecutar UP con el semáforo del dispositivo, de modo que en el paso 7 el planificador pueda ejecutar el administrador del dispositivo. Desde luego, si ahora varios procesos están listos, el planificador puede optar por ejecutar a continuación un proceso aún más importante. Más adelante en este capítulo veremos cómo se realiza la planificación.

En el ejemplo de la Fig. 2-12, realmente usamos los semáforos de dos formas distintas. Esta diferencia es lo bastante importante como para hacerla explícita. El semáforo mutex se usa para exclusión mutua; está diseñado para garantizar que sólo un proceso a la vez estará leyendo o escribiendo el buffer y las variables asociadas a él. Esta exclusión mutua es necesaria para evitar el caos.

El otro uso de los semáforos es la sincronización. Los semáforos full y empty se necesitan para garantizar que ciertas secuencias de sucesos ocurran o no ocurran. En este caso, los semáforos aseguran que el productor dejará de ejecutarse cuando el buffer esté lleno y que el consumidor dejará de ejecutarse cuando el buffer esté vacío. Este uso es diferente de la exclusión mutua. Aunque los semáforos se han usado desde hace más de un cuarto de siglo, todavía se siguen efectuando investigaciones sobre su uso. Por ejemplo, véase (Tai y Carver, 1996).

◆ Transferencia de mensajes

Esa otra cosa es la transferencia de mensajes. Este método de comunicación entre procesos utiliza dos primitivas SEND y RECEIVE que, al igual que los semáforos y a diferencia de los monitores, son llamadas al sistema y no construcciones del lenguaje. Como tales, es fácil colocarlas en procedimientos de biblioteca, como send (destino, &mensaje); y receive (origen, &mensaje);

La primera llamada envía un mensaje a un destino dado, y la segunda recibe un mensaje de un origen dado (o de cualquiera [ANY] si al receptor no le importa). Si no hay un mensaje disponible, el receptor podría bloquearse hasta que uno llegue. Como alternativa, podría regresar de inmediato con un código de error.

Aspectos de diseño de los sistemas de transferencia de mensajes

Los sistemas de transferencia de mensajes tienen muchos problemas y aspectos de diseño complicados que no se presentan con los semáforos ni con los monitores, sobre todo si los

procesos en comunicación están en diferentes máquinas conectadas por una red. Por ejemplo, se pueden perder mensajes en la red.

Para protegerse contra la pérdida de mensajes, el emisor y el receptor pueden convenir que, tan pronto como se reciba un mensaje, el receptor enviará de regreso un mensaje especial de acuse de recibo o confirmación. Si el emisor no recibe el acuse dentro de cierto intervalo de tiempo, retransmitirá el mensaje.

Consideremos ahora lo que sucede si el mensaje en sí se recibe correctamente, pero se pierde el acuse de recibo. El emisor retransmitirá el mensaje, de modo que el receptor lo recibirá dos veces. Es indispensable que el receptor pueda distinguir un mensaje nuevo de la retransmisión de uno viejo. Por lo regular, este problema se resuelve incluyendo números de secuencia consecutivos en cada mensaje original. Si el receptor recibe un mensaje que tiene el mismo número de secuencia que uno anterior, sabrá que el mensaje es un duplicado y podrá ignorarlo.

Los sistemas de mensajes también tienen que resolver la cuestión del nombre de los procesos, a fin de que el proceso especificado en una llamada SEND o RECEIVE no sea ambiguo. La verificación de autenticidad es otro problema en los sistemas de mensajes: ¿cómo puede el cliente saber que se está comunicando con el verdadero servidor de archivos, y no con un impostor?

En el otro extremo del espectro, hay aspectos de diseño que son importantes cuando el emisor y el receptor están en la misma máquina. Uno de éstos es el rendimiento. El copiado de mensajes de un proceso a otro siempre es más lento que efectuar una operación de semáforo o entrar en un monitor. Se ha trabajado mucho tratando de hacer eficiente la transferencia de mensajes. Cheriton (1984), por ejemplo, ha sugerido limitar el tamaño de los mensajes a lo que cabe en los registros de la máquina, y efectuar luego la transferencia de mensajes usando los registros.

3.1.3. El problema de productor-consumidor con transferencia de mensajes

Veamos ahora cómo puede resolverse el problema de productor-consumidor usando transferencia de mensajes y sin compartir memoria. En la Fig. 2-15 se presenta una solución. Suponemos que todos los mensajes tienen el mismo tamaño y que el sistema operativo coloca automáticamente en buffers los mensajes enviados pero aún no recibidos. En esta solución se usa un total de N mensajes, análogos a las N ranuras de un buffer en memoria compartida. El consumidor inicia enviando N mensajes vacíos al productor. Cada vez que el productor tiene un elemento que entregar al consumidor, toma un mensaje vacío y devuelve uno lleno. De este modo, el número total de mensajes en el sistema permanece constante y pueden almacenarse en una cantidad de memoria que se conoce con antelación.

Si el productor trabaja con mayor rapidez que el consumidor, todos los mensajes quedarán llenos, esperando al consumidor; el productor se bloqueará, esperando la llegada de un mensaje vacío. Si el consumidor trabaja con mayor rapidez, ocurre lo opuesto: todos los mensajes estarán vacíos esperando que el productor los llene; el consumidor estará bloqueado, esperando un mensaje lleno.

```
#define N 100                                /* número de ranuras del buffer */

void producer(void)
{
    int item;
    message m;                               /* buffer de mensaje */

    while (TRUE) {
        produce_item(&item);                 /* generar algo que poner en el buffer */
        receive(consumer, &m);               /* esperar que llegue un mensaje vacío */
        build_message(&m, item);             /* construir un mensaje para enviar */
        send(consumer, &m);                  /* enviar elemento al consumidor */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for(i = 0; i < N; i++) send(producer, &m); /* enviar N mensajes vacíos */
    while (TRUE) {
        receive(producer, &m);               /* obtener mensaje que contiene elemento */
        extract_item(&m, &item);             /* extraer elemento del mensaje */
        send(producer, &m);                  /* devolver una respuesta vacía */
        consume_item(item);                  /* hacer algo con el elemento */
    }
}
```

La transferencia de mensajes puede tener muchas variantes. Para comenzar, veamos cómo se dirigen los mensajes. Una forma es asignar a cada proceso una dirección única y hacer que los mensajes se dirijan a los procesos. Un método distinto consiste en inventar una nueva estructura de datos, llamada buzón.

Un buzón es un lugar donde se almacena temporalmente cierta cantidad de mensajes, que normalmente se especifican cuando se crea el buzón. Si se usan buzones, los parámetros de dirección de las llamadas SEND y RECEIVE son buzones, no procesos. Cuando un proceso trata de transmitir a un buzón que está lleno, queda suspendido hasta que se retira un mensaje de ese buzón, dejando espacio para uno nuevo. En el caso del problema de productor-consumidor, tanto el productor como el consumidor crearían buzones con espacio suficiente para N mensajes. El

productor enviaría mensajes con datos al buzón del consumidor, y éste enviaría mensajes vacíos al buzón del productor. Si se usan buzones, el mecanismo de almacenamiento temporal es claro: el buzón de destino contiene mensajes que se han enviado al proceso de destino pero todavía no han sido aceptados.

La otra forma extrema de manejar buzones es eliminar todo el almacenamiento temporal. Cuando se adopta este enfoque, si el SEND se ejecuta antes que el RECEIVE, el proceso emisor queda bloqueado hasta que ocurre el RECEIVE, y en ese momento el mensaje podrá copiarse directamente del emisor al receptor, sin buffers intermedios. De forma similar, si el RECEIVE se ejecuta primero, el receptor se bloquea hasta que ocurre el SEND. Esta estrategia se conoce como cita o rendezvous; es más fácil de implementar que un esquema de mensajes con almacenamiento temporal, pero es menos flexible, pues se obliga al emisor y al receptor a operar estrictamente sincronizados.

La comunicación entre los procesos de usuario en MINIX (y en UNIX) se efectúa a través de conductos, que efectivamente son buzones. La única diferencia real entre un sistema de mensajes con buzones y el mecanismo de conductos es que los conductos no preservan los límites de los mensajes. Dicho de otro modo, si un proceso escribe 10 mensajes de 100 bytes cada uno en un conducto y otro proceso lee 1000 bytes de ese conducto, el lector obtendrá los 10 mensajes a la vez. Con un verdadero sistema de mensajes, cada READ debería devolver sólo un mensaje. Desde luego, si los procesos convienen en leer y escribir siempre mensajes de tamaño fijo del conducto, o en terminar cada mensaje con un carácter especial (p. ej., salto de línea), no habrá problema. Los procesos que constituyen el sistema operativo MINIX mismo utilizan un verdadero esquema de mensajes de tamaño fijo para comunicarse entre sí.

◆ El problema de lectores y escritores

El problema de la cena de filósofos es útil para modelar procesos que compiten por tener acceso exclusivo a un número limitado de recursos, como dispositivos de E/S. Otro problema famoso es el de los lectores y escritores (Courtois et al., 1971), que modela el acceso a una base de datos.

Imaginemos, por ejemplo, un sistema de reservaciones de una línea aérea, con muchos procesos competidores que desean leerlo y escribir en él. Es aceptable tener múltiples procesos leyendo la base de datos al mismo tiempo, pero si un proceso está actualizando (escribiendo en) la base de datos, ningún otro podrá tener acceso a ella, ni siquiera los lectores. La pregunta es, ¿cómo programamos a los lectores y escritores? Una solución se muestra en la Fig. 2-19.

```
typedef int semaphore;           /* use su imaginación */
semaphore mutex = 1;           /* controla el acceso a 'rc' */
semaphore db = 1;              /* controla el acceso a la base de datos */
int rc = 0;                    /* núm. de procesos que leen o quieren leer */

void reader(void)
{
    while (TRUE) {              /* repetir indefinidamente */
        down(&mutex);           /* obtener acceso exclusivo a 'rc' */
        rc = rc + 1;            /* ahora un lector más */
        if (rc == 1) down(&db); /* si éste es el primer lector ... */
        up(&mutex);             /* liberar el acceso exclusivo a 'rc' */
        read_data_base();       /* acceder a los datos */
        down(&mutex);           /* obtener acceso exclusivo a 'rc' */
        rc = rc - 1;            /* ahora un lector menos */
        if (rc == 0) up(&db);    /* si éste es el último lector ... */
        up(&mutex);             /* liberar el acceso exclusivo a 'rc' */
        use_data_read();        /* región no crítica */
    }
}

void writer(void)
{
    while (TRUE) {              /* repetir indefinidamente */
        think_up_data();         /* región no crítica */
        down(&db);              /* obtener acceso exclusivo */
        write_data_base();       /* actualizar los datos */
        up(&db);                /* liberar el acceso exclusivo */
    }
}
```

Figura 2-19. Una solución al problema de lectores y escritores.

En esta solución, el primer lector que obtiene acceso a la base de datos ejecuta DOWN con el semáforo db. Los lectores subsecuentes se limitan a incrementar un contador, rc. Conforme los lectores salen, decrementan el contador, y el último en salir ejecuta UP con el semáforo para permitir que un escritor bloqueado, si lo había, entre.

La solución que presentamos aquí contiene implícitamente una sutil decisión que vale la pena comentar. Supongamos que mientras un lector está usando la base de datos, llega otro lector. Puesto que tener dos lectores al mismo tiempo no está prohibido, se admite al segundo lector. También pueden admitirse un tercer lector y lectores subsecuentes si llegan.

Supongamos ahora que llega un escritor. El escritor no puede ser admitido en la base de datos, pues requiere acceso exclusivo, de modo que el escritor queda suspendido. Más adelante, llegan lectores adicionales. En tanto haya al menos un lector activo, se admitirán lectores subsecuentes. A consecuencia de esta estrategia, en tanto haya un suministro constante de lectores, entrarán

tan pronto como lleguen. El escritor se mantendrá suspendido hasta que no haya ningún lector presente. Si llega un lector, digamos, cada 2 segundos, y cada lector tarda 5 segundos en efectuar su trabajo, el escritor nunca entrará.

Para evitar esta situación, el programa podría incluir una pequeña modificación: cuando llega un lector y un escritor está esperando, el lector queda suspendido detrás del escritor en lugar de ser admitido inmediatamente. Así, un escritor tiene que esperar hasta que terminan los lectores que estaban activos cuando llegó, pero no a que terminen los lectores que llegaron después de él. La desventaja de esta solución es que logra menor concurrencia y por tanto un menor rendimiento. Courtois et al., presentan una solución que confiere prioridad a los escritores. Si desea conocer los detalles, remítase a su artículo.

<http://laurel.datsi.fi.upm.es/docencia/asignaturas/sod>

Andrew S. Tanenbaum: "Sistemas Operativos Distribuidos", Prentice Hall (1996).

3.1.3.1 Planificación de Procesos

En los ejemplos de las secciones anteriores tuvimos varias situaciones en las que dos o más procesos (p. ej., productor y consumidor) podían ejecutarse lógicamente. Cuando hay más de un proceso ejecutable, el sistema operativo debe decidir cuál ejecutará primero. La parte del sistema operativo que toma esta decisión se denomina planificador; el algoritmo que usa se denomina algoritmo de planificación.

En la época de los sistemas por lote con entradas en forma de imágenes de tarjetas en una cinta magnética, el algoritmo de planificación era sencillo: simplemente se ejecutaba el siguiente trabajo de la cinta. En los sistemas de tiempo compartido, el algoritmo de planificación es más complejo, pues es común que haya varios usuarios en espera de ser atendidos, y también puede haber uno o más flujos por lotes (p. ej., en una compañía de seguros, para procesar reclamaciones). Incluso en las computadoras personales, puede haber varios procesos iniciados por el usuario compitiendo por la CPU, sin mencionar los trabajos de segundo plano, como los demonios de red o de correo electrónico que envían o reciben mensajes.

Antes de examinar algoritmos de planificación específicos, debemos pensar en qué está tratando de lograr el planificador. Después de todo, éste se ocupa de decidir una política, no de proveer un mecanismo. Se nos ocurren varios criterios para determinar en qué consiste un buen algoritmo de planificación. Entre las posibilidades están:

1. Equitatividad —asegurarse de que cada proceso reciba una parte justa del tiempo de CPU.
2. Eficiencia —mantener la CPU ocupada todo el tiempo.

3. Tiempo de respuesta —minimizar el tiempo de respuesta para usuarios interactivos.
4. Retorno —minimizar el tiempo que los usuarios por lotes tienen que esperar sus salidas.
5. Volumen de producción —maximizar el número de trabajos procesados por hora.

Si pensamos un poco veremos que algunos de estos objetivos son contradictorios. Si queremos minimizar el tiempo de respuesta para los usuarios interactivos, el planificador no deberá ejecutar trabajos por lotes (excepto quizá entre las 3 A.M. y las 6 A.M., cuando todos los usuarios interactivos están muy a gusto en sus camas). A los usuarios por lotes seguramente no les gustaría este algoritmo, pues viola el criterio 4.

Puede demostrarse (Kleinrock, 1975) que cualquier algoritmo de planificación que dé preferencia a una clase de trabajos perjudicará a los de otras clases. Después de todo, la cantidad de tiempo de CPU disponible es finita. Para darle más a un usuario tenemos que darle menos a otro. Así es la vida.

Una complicación que deben enfrentar los planificadores es que cada proceso es único e impredecible. Algunos dedican una buena parte del tiempo a esperar E/S de archivos, mientras otros usarían la CPU durante horas si se les permitiera hacerlo. Cuando el planificador comienza a ejecutar un proceso, nunca sabe con certeza cuánto tiempo pasará antes de que dicho proceso se bloquee, sea para E/S, en espera de un semáforo o por alguna otra razón. Para asegurarse de que ningún proceso se ejecute durante demasiado tiempo, casi todas las computadoras tienen incorporado un cronómetro o reloj electrónico que genera interrupciones periódicamente.

Es común que la frecuencia sea de 50060 interrupciones por segundo (equivalente a 50 o 60 hertz, abreviado Hz), pero en muchas computadoras el sistema operativo puede ajustar la frecuencia del cronómetro al valor que desee. En cada interrupción de reloj, el sistema operativo se ejecuta y decide si debe permitirse que el proceso que se está ejecutando actualmente continúe o si ya disfrutó de suficiente tiempo de CPU por el momento y debe suspenderse para otorgar a otro proceso la CPU.

La estrategia de permitir que procesos lógicamente ejecutables se suspendan temporalmente se denomina planificación expropiativa y contrasta con el método de ejecución hasta terminar de los primeros sistemas por lotes. La ejecución hasta terminar también se denomina planificación no expropiativa. Como hemos visto a lo largo del capítulo, un proceso puede ser suspendido en un instante arbitrario, sin advertencia, para que otro proceso pueda ejecutarse. Esto da pie a condiciones de competencia y requiere semáforos, monitores, mensajes o algún otro método avanzado para prevenirlas.

Por otro lado, una política de dejar que los procesos se ejecuten durante el tiempo que quieran implicaría que un proceso que está calculando π con una precisión de mil millones de cifras podría privar de servicio a todos los demás procesos indefinidamente.

Así, aunque los algoritmos de planificación no apropiativos son sencillos y fáciles de implementar, por lo regular no son apropiados para sistemas de aplicación general con varios usuarios que compiten entre sí. Por otro lado, en un sistema dedicado como un servidor de base de datos, bien puede ser razonable que el proceso padre inicie un proceso hijo para trabajar con una solicitud y dejarlo que se ejecute hasta terminar o bloquearse. La diferencia respecto al sistema de aplicación general es que todos los procesos del sistema de bases de datos están bajo el control de un solo amo, que sabe lo que cada hijo va a hacer y cuánto va a tardar.

◆ Planificación round robin (de torneo)

Examinemos ahora algunos algoritmos de planificación específicos. Uno de los más antiguos, sencillos, equitativos y ampliamente utilizados es el de round robin. A cada proceso se le asigna un intervalo de tiempo, llamado cuanto, durante el cual se le permite ejecutarse. Si el proceso todavía se está ejecutando al expirar su cuanto, el sistema operativo se apropia de la CPU y se la da a otro proceso. Si el proceso se bloquea o termina antes de expirar el cuanto, la conmutación de CPU naturalmente se efectúa cuando el proceso se bloquee. El round robin es fácil de implementar. Todo lo que el planificador tiene que hacer es mantener una lista de procesos ejecutables, como se muestra en la Fig. 2-22(a). Cuando un proceso gasta su cuanto, se le coloca al final de la lista, como se aprecia en la Fig. 2-22(b).

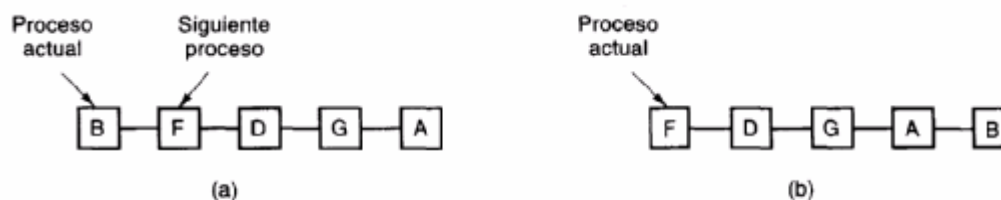


Figura 2-22. Planificación *round robin*. (a) La lista de procesos ejecutables. (b) La lista de procesos ejecutables después de que *B* gasta su cuanto.

La única cuestión interesante cuando se usa el round robin es la duración del cuánto. La conmutación de un proceso a otro requiere cierto tiempo para llevar a cabo las tareas administrativas: guardar y cargar registros y mapas de memoria, actualizar diversas tablas y listas, etc. Su pongamos que esta conmutación de proceso o conmutación de contexto requiere 5 ms. Supongamos también que usamos cuantos de 20 ms. Con estos parámetros, después de realizar trabajo útil durante 20 ms, la CPU tendrá que ocupar 5 ms en la conmutación de procesos. Se desperdiciará el 20% del tiempo de CPU en gastos extra administrativos.

A fin de mejorar la eficiencia de la CPU, podríamos usar cuantos de, digamos, 500 ms. Ahora el tiempo desperdiciado es de menos del 1%, pero consideremos lo que sucede en un sistema de tiempo compartido si 10 usuarios interactivos pulsán la tecla de retorno de carro aproximadamente al mismo tiempo: diez procesos se pondrían en la lista de procesos ejecutables. Si la CPU está ociosa, el primero se iniciará de inmediato, el segundo podría no iniciarse hasta cerca de medio segundo después, y así sucesivamente. El pobre proceso que le haya tocado ser último podría tener que esperar 5 segundos antes de tener su oportunidad, suponiendo que los demás procesos utilizan su cuanto completo. Para casi cualquier usuario, un retardo de 5 segundos en la respuesta a un comando corto sería terrible. El mismo problema puede presentarse en una computadora personal que maneja multiprogramación.

La conclusión puede formularse así: escoger un cuanto demasiado corto causa demasiadas conmutaciones de procesos y reduce la eficiencia de la CPU, pero escogerlo demasiado largo puede dar pie a una respuesta deficiente a solicitudes interactivas cortas. Un cuanto de cerca de 100 ms suele ser un término medio razonable.

◆ Planificación por prioridad

La planificación en round robin supone implícitamente que todos los procesos son igualmente importantes. Con frecuencia, las personas que poseen y operan sistemas de computadora multiusuario tienen ideas diferentes acerca del tema. En una universidad, la jerarquía puede consistir en decanos primero, luego profesores, secretarías, conserjes y, por último, estudiantes. La necesidad de tener en cuenta factores externos da pie a la planificación por prioridad. La idea básica es sencilla: a cada proceso se le asigna una prioridad, y se permite que se ejecute el proceso ejecutable que tenga la prioridad más alta.

Incluso en una PC con un solo dueño, puede haber múltiples procesos, algunos más importantes que otros. Por ejemplo, un proceso demonio que envía correo electrónico en segundo plano debe tener menor prioridad que un proceso que está exhibiendo video en tiempo real en la pantalla.

A fin de evitar que los procesos de alta prioridad se ejecuten indefinidamente, el planificador puede reducir la prioridad de los procesos que actualmente se ejecutan en cada tic del reloj (esto es, en cada interrupción de reloj). Si esta acción hace que la prioridad se vuelva menor que la del siguiente proceso con más alta prioridad, ocurrirá una conmutación de procesos. Como alternativa, se podría asignar a cada proceso un cuanto máximo en el que se le permitiera tener la CPU continuamente; cuando se agota este cuanto, se da oportunidad al proceso con la siguiente prioridad más alta de ejecutarse.

Podemos asignar prioridades a los procesos estática o dinámicamente. En una computadora militar, los procesos iniciados por generales podrían comenzar con prioridad 100, los iniciados por

coroneles con 90, por mayores con 80, por capitanes con 70, por tenientes con 60, etc. Como alternativa, en un centro de cómputo comercial, los procesos de alta prioridad podrían costar 100 dólares por hora, los de mediana prioridad 75 dólares por hora, y los de baja prioridad 50 dólares por hora. El sistema UNIX tiene un comando, fice, que permite a un usuario reducir voluntariamente la prioridad de su proceso, con objeto de ser amable con los demás usuarios. Nadie lo usa.

El sistema también puede asignar prioridades dinámicamente a fin de lograr ciertos objetivos del sistema. Por ejemplo, algunos procesos están limitados principalmente por E/S y pasan la mayor parte del tiempo esperando que terminen operaciones de BIS. Siempre que un proceso necesita la CPU, se le deberá otorgar de inmediato, con objeto de que pueda iniciar su siguiente solicitud de E/S, que entonces podrá proceder en paralelo con otro proceso que sí está realizando cálculos. Si hiciéramos que los procesos limitados por BIS esperaran mucho tiempo la CPU, implicaría tenerlo por ahí ocupando memoria durante un tiempo innecesariamente largo. Un algoritmo sencillo para dar buen servicio a los procesos limitados por E/S es asignarles la prioridad $1/f$, donde f es la fracción del último cuanto que un proceso utilizó. Un proceso que usó sólo 2 ms de su cuanto de 100 ms recibiría una prioridad de 50, en tanto que un proceso que se ejecutó 50 ms antes de bloquearse obtendría una prioridad de 2, y uno que ocupó todo su cuanto obtendría una prioridad de 1.

En muchos casos es conveniente agrupar los procesos en clases de prioridad y usar planificación por prioridad entre las clases pero planificación round robin dentro de cada clase. La Fig. 2-23 muestra un sistema con cuatro clases de prioridad. El algoritmo de planificación es el siguiente: en tanto haya procesos ejecutables en la clase de prioridad 4, se ejecutará cada uno durante un cuanto, con round robin, sin ocuparse de las clases de menor prioridad. Si la clase de prioridad 4 está vacía, se ejecutan los procesos de la clase 3 con round robin. Si tanto la clase 4 como la 3 están vacías, se ejecutan los procesos de clase 2 con round robin, etc. Si las prioridades no se ajustan ocasionalmente, las clases de baja prioridad podrían morir de inanición.

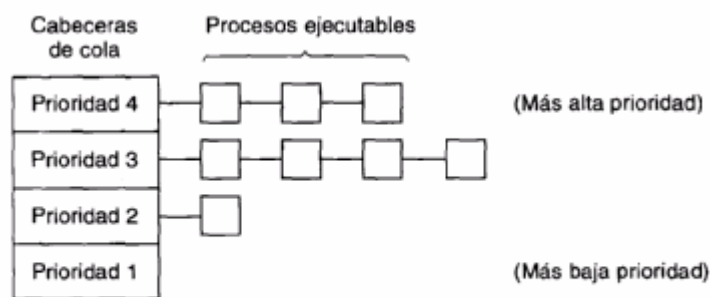


Figura 2-23. Algoritmo de planificación con cuatro clases de prioridad.

◆ Colas múltiples

Uno de los primeros planificadores por prioridad se incluyó en CTSS (Corbato et al., 1962). CTSS tenía el problema de que la conmutación de procesos era muy lenta porque la 7094 sólo podía contener un proceso en la memoria. Cada conmutación implicaba escribir el proceso actual en disco y leer uno nuevo del disco.

Los diseñadores de CTSS pronto se dieron cuenta de que resultaba más eficiente dar a los procesos limitados por CPU un cuanto largo de vez en cuando, en lugar de darles cuantos pequeños muy a menudo (porque se reducía el intercambio). Por otro lado, dar a todos los procesos un cuanto largo implicaría un tiempo de respuesta deficiente, como ya hemos visto.

Su solución consistió en establecer clases de prioridad. Los procesos de la clase más alta se ejecutaban durante un cuánto. Los procesos de la siguiente clase más alta se ejecutaban durante dos cuantos. Los procesos de la siguiente clase se ejecutaban durante cuatro cuantos, y así sucesivamente. Cada vez que un proceso agotaba todos los cuantos que tenía asignados, se le degradaba una clase.

Por ejemplo, consideremos un proceso que necesita calcular continuamente durante 100 cuantos; inicialmente, se le daría un cuanto, y luego se intercambiaría por otro proceso. La siguiente vez, recibiría dos cuantos antes de ser intercambiado. En ocasiones subsecuentes obtendría 4, 8, 16, 32 y 64 cuantos, aunque sólo usaría 37 de los últimos 64 cuantos para completar su trabajo. Sólo se necesitarían 7 intercambios (incluida la carga inicial) en lugar de 100 si se usara un algoritmo round robin puro. Además, al hundirse el proceso progresivamente en las colas de prioridad, se le ejecutaría cada vez con menor frecuencia, guardando la CPU para procesos interactivos cortos.

Se adoptó la siguiente política para evitar que un proceso que en el momento de iniciarse necesita ejecutarse durante un tiempo largo pero posteriormente se vuelve interactivo fuera castigado indefinidamente. Cada vez que en una terminal se pulsaba el retorno de carro, el proceso perteneciente a esa terminal se pasaba a la clase de más alta prioridad, bajo el supuesto de que estaba a punto de volverse interactivo. Un buen día un usuario con un proceso muy limitado por CPU descubrió que si se sentaba ante su terminal y pulsaba el retorno de carro al azar cada varios segundos su tiempo de respuesta mejoraba notablemente. Este usuario se lo dijo a todos sus amigos. Moraleja de la historia: acertar en la práctica es mucho más difícil que acertar en la teoría.

Se han utilizado muchos otros algoritmos para asignar procesos a clases de prioridad. Por ejemplo, el influyente sistema XDS 940 (Lampson, 1968), construido en Berkeley, tenía cuatro clases de prioridad, llamadas terminal, E/S, cuanto corto y cuanto largo. Cuando un proceso que estaba esperando entradas de la terminal finalmente se despertaba, pasaba a la clase de prioridad más alta (terminal). Cuando un proceso que estaba esperando un bloque de disco quedaba listo,

pasaba a la segunda clase. Si un proceso seguía en ejecución en el momento de expirar su cuanto, se le colocaba inicialmente en la tercera clase, pero si agotaba su cuanto demasiadas veces seguidas sin bloquearse para E/S de terminal o de otro tipo, se le bajaba a la cuarta cola. Muchos otros sistemas usan algo similar para dar preferencia a los usuarios y procesos interactivos por encima de los de segundo plano.

◆ El primer trabajo más corto

La mayor parte de los algoritmos anteriores se diseñaron para sistemas interactivos. Examinemos ahora uno que resulta especialmente apropiado para los trabajos por lotes cuyos tiempos de ejecución se conocen por adelantado. En una compañía de seguros, por ejemplo, es posible predecir con gran exactitud cuánto tiempo tomará ejecutar un lote de 1000 reclamaciones, pues se efectúan trabajos similares todos los días. Si hay varios trabajos de igual importancia esperando en la cola de entrada para ser iniciados, el planificador deberá usar el criterio del primer trabajo más corto. Examinemos la Fig. 2-24. Aquí encontramos cuatro trabajos, A, B, C y D, con tiempos de ejecución de 8, 4, 4 y 4 minutos, respectivamente. Si los ejecutamos en ese orden, el tiempo de retomo para A será de 8 minutos, para B, de 12 minutos, para C, de 16 minutos, y para D, de 20 minutos, siendo el promedio de 14 minutos.



Figura 2-24. Ejemplo de planificación del primer trabajo más corto.

Consideremos ahora la ejecución de estos trabajos usando el primer trabajo más corto, como se muestra en la Fig. 2-24(b). Los tiempos de retomo son ahora de 4, 8, 12 y 20 minutos para un promedio de 11 minutos. Se puede demostrar que la política del primer trabajo más corto es óptima. Consideremos el caso de cuatro trabajos, con tiempos de ejecución de a, b, c y d, respectivamente. El primer trabajo termina en un tiempo a, el segundo, en a + b, etc. El tiempo de retomo medio es $(4a + 3b + 2c + d)/4$. Es evidente que a contribuye más al promedio que los demás tiempos, por lo que debe ser el trabajo más corto, siguiendo b, c y por último d, que es el más largo y sólo afecta su propio tiempo de retomo. El mismo argumento es aplicable a cualquier cantidad de trabajos.

Dado que la política del primer trabajo más corto produce el tiempo de respuesta medio mínimo, sería deseable poderlo usar también para procesos interactivos. Esto es posible hasta cierto punto. Los procesos interactivos generalmente siguen el patrón de esperar un comando, ejecutar el comando, esperar un comando, ejecutar el comando, etc. Si consideramos la ejecución de cada

comando como un “trabajo” individual, podremos minimizar el tiempo de respuesta global ejecutando primero el trabajo más corto. El único problema es determinar cuál de los procesos ejecutables es el más corto.

Una estrategia consiste en hacer estimaciones basadas en el comportamiento histórico y ejecutar el proceso con el tiempo de ejecución estimado más corto. Supongamos que el tiempo por comando estimado para cierta terminal es T_0 . Supongamos ahora que se mide su siguiente ejecución, dando T_1 . Podríamos actualizar nuestro estimado calculando una suma ponderada de estos dos números, es decir, $aT_0 + (1 - a)T_1$. Dependiendo del valor que escojamos para a , podremos hacer que el proceso de estimación olvide las ejecuciones viejas rápidamente, o las recuerde durante mucho tiempo. Con $a = 1/2$, obtenemos estimaciones sucesivas de T_0 , $T_0/2 + T_1/2$, $T_0/4 + T_1/4 + T_2/2$, $T_0/8 + T_1/8 + T_2/4 + T_3/2$.

Después de tres nuevas ejecuciones, el peso de T_0 en el nuevo estimado se ha reducido a $1/8$. La técnica de estimar el siguiente valor de una serie calculando la media ponderada del valor medido actual y el estimado previo también se conoce como maduración, y es aplicable a muchas situaciones en las que debe hacerse una predicción basada en valores previos. La maduración es especialmente fácil de implementar cuando $a = 1/2$. Todo lo que se necesita es sumar el nuevo valor al estimado actual y dividir la suma entre 2 (desplazándola a la derecha un bit).

Vale la pena señalar que el algoritmo del primer trabajo más corto sólo es óptimo cuando todos los trabajos están disponibles simultáneamente. Como contraejemplo, consideremos cinco trabajos, A a E, con tiempos de ejecución de 2, 4, 1, 1 y 1, respectivamente. Sus tiempos de llegada son 0, 0, 3, 3 y 3.

Inicialmente, sólo pueden escogerse A o B, puesto que los otros tres trabajos todavía no llegan. Si ejecutamos el primer trabajo más corto, seguiremos el orden de ejecución A, B, C, D, E logrando una espera media de 4.6. Sin embargo, si los ejecutamos en el orden B, C, D, E y A la espera media será de 4.4.

◆ Planificación de dos niveles

Hasta ahora más o menos hemos supuesto que todos los procesos ejecutables están en la memoria principal. Si la memoria principal disponible no es suficiente, algunos de los procesos ejecutables tendrán que mantenerse en el disco total o parcialmente. Esta situación tiene implicaciones importantes para la planificación, ya que el tiempo de conmutación de procesos cuando hay que traer los procesos del disco es varios órdenes de magnitud mayor que cuando la conmutación es a un proceso que ya está en la memoria.

Una forma más práctica de manejar los procesos intercambiados a disco es el uso de un planificador de dos niveles. Primero se carga en la memoria principal un subconjunto de los

procesos ejecutables, como se muestra en la Fig. 2-25(a). Luego, el planificador se limita a escoger procesos de este subconjunto durante cierto tiempo. Periódicamente se invoca un planificador de nivel superior para eliminar los procesos que han estado en memoria suficiente tiempo y cargar procesos que han estado en el disco demasiado tiempo. Una vez efectuado el cambio, como en la Fig. 2-25(b), el planificador de bajo nivel otra vez se limita a ejecutar procesos que están en la memoria. Así, este planificador se ocupa de escoger entre los procesos ejecutables que están en la memoria en ese momento, mientras el planificador de nivel superior se ocupa de trasladar procesos entre la memoria y el disco.

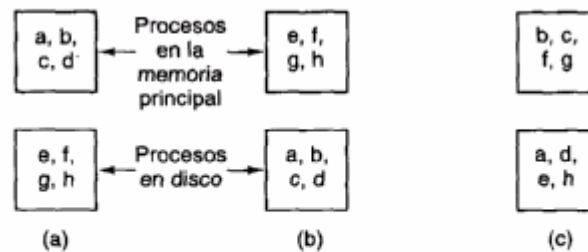


Figura 2-25. Un planificador de dos niveles debe transferir procesos entre el disco y la memoria y también escoger los procesos a ejecutar de entre los que están en la memoria. Representamos tres diferentes instantes con (a), (b) y (c).

Entre los criterios que el planificador de nivel superior podría usar para tomar sus decisiones están los siguientes:

1. ¿Cuánto tiempo hace que el proceso se intercambió del o al disco?
2. ¿Cuánto tiempo de CPU ha recibido el proceso recientemente?
3. ¿Qué tan grande es el proceso? (Los pequeños no estorban.)
4. ¿Qué tan alta es la prioridad del proceso?

Aquí también podríamos usar planificación round robín, por prioridad o por cualquiera de varios otros métodos. Los dos planificadores podrían usar el mismo algoritmo o algoritmos distintos.

<http://laurel.datsi.fi.upm.es/docencia/asignaturas/sod>

Andrew S. Tanenbaum: "Sistemas Operativos Distribuidos", Prentice Hall (1996).

3.2. Ejercicios por temas

◆ Ejercicio del tema 1

Definición de gestión de procesos

1. Explique colocando un ejemplo de cada situación:
 - a. El Pid es usado por el sistema operativo para:
 - b. Son iguales todos los contadores de programa de los procesos?
 - c. El segmento de datos de los procesos se puede explicar cómo:
 - d. Explique una interrupción como proceso asociado al sistema operativo
 - e. Una transición de un estado de disco a la ram se puede dar por:
2. Explique en qué se diferencia el control de hilos del control de procesos

◆ Ejercicio del tema 2

Comunicación entre procesos y concurrencia

1. Cómo funcionan las llamadas al sistema Signal-Wait
2. En que consiste la concurrencia entre procesos y que puede ocasionar
3. Que quiere decir que un proceso este dentro de una región critica
4. De un ejemplo de memoria compartida entre procesos
5. El principio de exclusión mutua entre procesos se da para solucionar que tipo de problema entre los procesos.
6. Las soluciones de exclusión mutua permiten a los procesos?
7. La solución de semáforos basa su funcionamiento en?
8. Las transferencia de mensajes permite facilitar la exclusión mutua por?
9. En el problema del lector y escritor el semáforo bd sirve para?
10. La directiva send en la solución de transferencia de mensajes sirve para?

◆ Ejercicio del tema 3

Planificación de procesos

1. Suponga cinco procesos cargados en Ram, suponiendo que se pueden manejar

Prioridades para su planificación conteste:

1. Como se planificaría usando el algoritmo de Round Robin
2. Como se planificaría usando prioridades
3. Como se planificaría usando el tiempo más corto
4. Compare el algoritmo de round robin con el de colas múltiples de prioridades
5. Explique de forma gráfica como se da la planificación en dos niveles

Prueba Final

Para los siguientes enunciados seleccione la respuesta acertada a cada pregunta propuesta:

1. Si el cp de un proceso está apuntando a la instrucción x después de su ejecución y es vuelto a llamar por el sistema operativo seguirá con:
 - a. La instrucción x
 - b. Instrucción x-1
 - c. Instrucción x+1
 - d. Instrucción x+2
2. Si un proceso en ejecución le falta un dato que se lee por el teclado debe pasar al estado:
 - a. Listo
 - b. Bloqueado
 - c. Terminar
 - d. Ejecución
3. Un proceso en listo en el disco duro puede pasar a la ram si:
 - a. Si necesita ejecutarse
 - b. Si hace transición por los estados del disco
 - c. Si es intercambiado con uno de memoria
 - d. Si va directamente a ejecución
4. El principio de exclusión mutua explica que:
 - a. Dos procesos pueden estar en una región critica
 - b. Un proceso que ingresa y sale de una región critica deja a otro esperando fuera de ella
 - c. Los procesos nunca pueden entrar a regiones criticas
 - d. Que un proceso que entra deja a otro fuera de la región critica

5. La operación down asociada a un semáforo permite:
 - a. Que un proceso entre a una región crítica o se bloquee
 - b. Que un proceso salga o desbloquee a otro
 - c. Que un proceso entre o desbloquee a otro
 - d. Que un procesos salga o bloqueen otros
6. La exclusión mutua con semáforos funciona a partir de:
 - a. Las llamadas send y receive
 - b. Las operaciones dormir y despertar
 - c. Las operaciones down y up
 - d. Una variable de cerradura
7. La planificación de procesos de primero el tiempo más corto tiene como filosofía planificar:
 - a. por prioridad
 - b. por tiempos de llegada
 - c. por colas de prioridades
 - d. por la tarea más corta

3.2.1. Actividad

1. Simular el algoritmo del lector y escritor con semáforos usando uno de los lenguajes actuales de alto nivel (Java).
2. Simular el comportamiento de los algoritmos de planificación de procesos.

4. ADMINISTRADOR DE MEMORIA

OBJETIVO GENERAL

- ◆ Conocer cómo se lleva el registro de memoria, para la determinación de la administración de su espacio cuando se hace ubicación de los procesos que se cargan o salida de procesos que terminan y la optimización de su utilización.

OBJETIVOS ESPECÍFICOS

- ◆ Conocer los distintos esquemas de administración de memoria a través del desarrollo histórico de los sistemas operativos.
- ◆ Entender como el sistema operativo lleva el registro de memoria
- ◆ Entender la relación entre el intercambio del disco y la ram como principio de la memoria virtual
- ◆ Conocer las técnicas principales de la memoria virtual
- ◆ Comprender la paginación y la segmentación para el mejoramiento de los principios de memoria virtual

Prueba Inicial

1. Para que se realiza administración de memoria por parte de un sistema operativo
2. Que se entiende por fragmentación de memoria
3. En qué casos se aplica la compactación de memoria
4. La reubicación y protección se pueden entender como problemas que afronta la gestión de RAM
5. Un proceso puede ejecutarse en un espacio de RAM inferior a su tamaño

4.1. Administración de la memoria sin intercambio o paginación

<http://laurel.datsi.fi.upm.es/docencia/asignaturas/sod>

Andrew S. Tanenbaum: "Sistemas Operativos Distribuidos", Prentice Hall (1996).

4.1.1. Administración Básica de Memoria

Los sistemas de administración de memoria se pueden dividir en dos clases, los que trasladan procesos entre la memoria y el disco durante la ejecución (intercambio y paginación) y los que no lo hacen. Estos últimos son más sencillos, así que los estudiaremos primero. Más adelante en el capítulo examinaremos el intercambio y la paginación. A lo largo de todo este capítulo, el lector debe tener presente que el intercambio y la paginación son en buena medida situaciones causadas por la falta de suficiente memoria principal para contener todos los programas a la vez. Al bajar el costo de la memoria principal, los argumentos a favor de un tipo de esquema de administración de memoria u otro pueden hacerse obsoletos, a menos que los programas crezcan con mayor rapidez que las memorias.

4.1.1.1 Monoprogramación sin Intercambio si Paginación

El esquema de administración de memoria más sencillo posible es ejecutar sólo un programa a la vez, compartiendo la memoria entre ese programa y el sistema operativo. En la Fig. 4-1 se muestran tres variaciones sobre este tema. El sistema operativo puede estar en la base de la memoria en RAM (memoria de acceso aleatorio), como se muestra en la Fig. 4-1 (a), o puede estar en ROM (memoria sólo de lectura) en la parte superior de la memoria, como en la Fig. 4-1(b), o lo» controladores de dispositivos pueden estar en la parte superior de la memoria en una ROM con el resto del sistema en RAM hasta abajo, como se muestra en la Fig. 4-1(c). Este último modelo es utilizado por los sistemas MS-DOS pequeños, por ejemplo. En las IBM PC, la porción del sistema que está en ROM se llama BIOS (Basic Input Output System, sistema básico de entrada salida).



Figura 4-1. Tres formas sencillas de organizar la memoria con un sistema operativo y un proceso de usuario. También existen otras posibilidades.

Si el sistema está organizado de esta manera, sólo puede ejecutarse un proceso a la vez. Tan pronto como el usuario teclea un comando, el sistema operativo copia el programa solicitador disco a la memoria y lo ejecuta. Cuando el proceso termina, el sistema operativo exhibe un carácter de indicación y espera un nuevo comando. Cuando el sistema operativo lo recibe, carga un nuevo programa en la memoria, sobre escribiendo el primero.

Multiprogramación con particiones fijas

Aunque a veces se usa la monoprogramación en computadoras pequeñas con sistemas operativos sencillos, hay muchos casos en que es deseable permitir la ejecución de múltiples procesos a la vez. En los sistemas de tiempo compartido, tener varios procesos en la memoria a la vez implica que cuando un proceso está bloqueado esperando que termine una E/S, otro puede usar la CPU. Así, la multiprogramación aumenta el aprovechamiento de la CPU. Sin embargo, incluso en computadoras personales hay ocasiones en las que resulta útil poder ejecutar dos o más programas a la vez.

La forma más fácil de lograr la multiprogramación consiste simplemente en dividir la memoria en n particiones, posiblemente desiguales. Esta división puede, por ejemplo, efectuarse manualmente cuando se inicia el sistema.

Cuando llega un trabajo, se le puede colocar en la cola de entrada de la partición pequeña que puede contenerlo. Puesto que las particiones están fijas en este esquema, cualquier espacio de una partición que un trabajo no utilice se desperdiciará. En la Fig. 4-2(a) vemos el aspecto que tiene este sistema de particiones fijas y colas de entrada individuales.

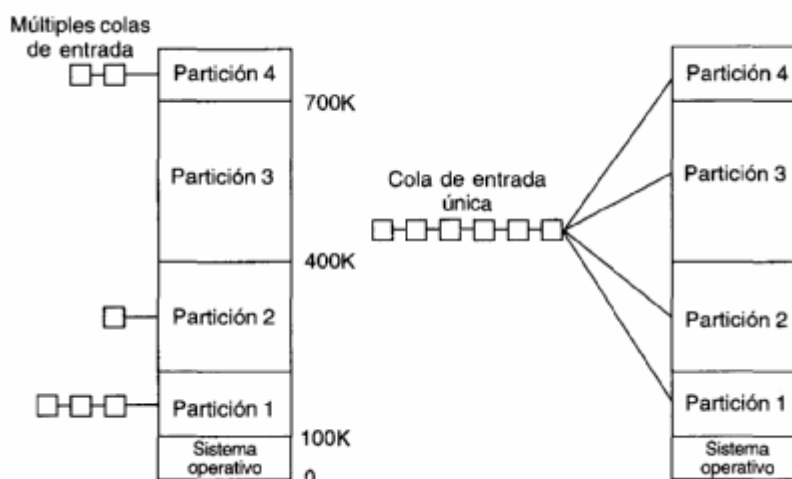


Figura 4-2. (a) Particiones de memoria fijas con colas de entrada individuales para cada partición, (b) Particiones de memoria fija con una sola cola de entrada.

La desventaja de repartir los trabajos entrantes en colas distintas se hace evidente cuando la cola de una partición grande está vacía pero la cola de una partición pequeña está llena, como es el caso de las particiones 1 y 3 en la Fig. 4-2(a). Una organización alternativa sería mantener una sola cola, como en la Fig. 4-2(b). Cada vez que se libera una partición, se selecciona el trabajo más cercano a la cabeza de la cola que cabe en esa partición, se carga en dicha partición y ejecuta. Puesto que no es deseable desperdiciar una partición grande en un trabajo pequeño, una estrategia diferente consiste en examinar toda la cola de entrada cada vez que se libera una partición y escoger el trabajo más grande que cabe en ella. Observe que este último algoritmo discrimina contra los trabajos pequeños, considerándolos indignos de recibir toda la partición, en tanto que usualmente es deseable dar el mejor servicio, y no el peor, a los trabajos más pequeños (que supuestamente son interactivos).

Una salida consiste en tener por lo menos una partición pequeña disponible. Una partición a permitirá que se ejecuten los trabajos pequeños sin tener que asignar una partición grande para ello. Otro enfoque sería adoptar la regla de que un trabajo que es elegible para ejecutarse no puede pasarse por alto más de k veces. Cada vez que se pase por alto, el trabajo recibirá un punto, cuando haya adquirido k puntos, no se le podrá pasar por alto otra vez.

Este sistema, con particiones fijas establecidas por el operador en la mañana y que no se modificaban posteriormente, fue utilizado por OS/360 en macrocomputadoras de IBM durante muchos años. Se le llamaba MFT (multiprogramación con un número fijo de tareas, u OS/MFT). Este sistema es fácil de entender e igualmente sencillo de implementar: los trabajos entrantes se ponen en cola hasta que está disponible una partición apropiada. En ese momento, el trabajo se

carga en esa partición y se ejecuta hasta terminar. Hoy día son pocos los sistemas operativos que dan soporte a este modelo, si es que todavía existe alguno.

◆ Relocalización y protección

La multiprogramación introduce dos problemas esenciales que es preciso resolver: la relocalización y la protección. Examinemos la Fig. 4-2. Es evidente que diferentes trabajos se ejecutarán en diferentes direcciones. Cuando se vincula un programa (es decir, cuando el programa principal, los procedimientos escritos por el usuario y los procedimientos de biblioteca se combinan en mí solo espacio de direcciones), el vinculador necesita saber en qué dirección de la memoria va a comenzar el programa.

Por ejemplo, supongamos que la primera instrucción es una llamada a un procedimiento que está en la dirección absoluta 100 dentro del archivo binario producido por el vinculador. Si este programa se carga en la partición 1, esa instrucción saltará a la dirección absoluta 100, que está, dentro del sistema operativo. Lo que se necesita es una llamada a $100K + 100$. Si el programa se le carga en la partición 2, deberá efectuarse como una llamada a $200K + 100$, etc. Este problema se denomina problema de relocalización.

Una posible solución es modificar realmente las instrucciones en el momento en que el programa se carga en la memoria. Se suma $100K$ a cada una de las direcciones de un programa que se carga en la partición 1, se suma $200K$ a las direcciones de los programas cargados en la partición 2, etc. A fin de efectuar la relocalización durante la carga de esta manera, el vinculador debe incluir en el programa binario una lista o mapa de bits que indique cuáles palabras del programa son direcciones que deben relocalizarse y cuáles con códigos de operación, constantes u otros elementos que no deben relocalizarse. OS/MFT funcionaba de este modo. Algunas microcomputadoras también trabajan así.

La relocalización durante la carga no resuelve el problema de la protección. Un programa mal intencionado siempre puede construir una nueva instrucción y saltar a ella. Dado que los programas en este sistema usan direcciones de memoria absolutas en lugar de direcciones relativas a un registro, no hay forma de impedir que un programa construya una instrucción que lea o escriba cualquier palabra de la memoria. En los sistemas multiusuario, no es conveniente permitir que los procesos lean y escriban en memoria que pertenece a otros usuarios.

La solución que IBM escogió para proteger el 360 fue dividir la memoria en bloques de 2K bytes y asignar un código de protección de cuatro bits a cada bloque. La PSW contenía una clave de cuatro bits. El hardware del 360 atrapaba cualquier intento, por parte de un proceso en ejecución, de acceder a memoria cuyo código de protección difería de la clave de la PSW. Puesto que sólo el sistema operativo podía cambiar los códigos de protección y la clave, los procesos de usuario no podían interferirse ni interferir el sistema operativo mismo.

Una solución alternativa tanto al problema de relocalización como al de protección consiste en equipar la máquina con dos registros especiales en hardware, llamados registros de base y de límite. Cuando se calendariza un proceso, el registro de base se carga con la dirección del principio de su partición, y el registro de límite se carga con la longitud de la partición. A cada dirección de memoria generada se le suma automáticamente el contenido del registro de base antes de enviarse a la memoria. De este modo, si el registro de base es de 100K, una instrucción CALL 100 se convierte efectivamente en una instrucción CALL 100K + 100, sin que la instrucción en sí se modifique. Además, las direcciones se cotejan con el registro de límite para asegurar que no intenten acceder a memoria fuera de la partición actual. El hardware protege los registros de base y de límite para evitar que los programas de usuario los modifiquen.

La CDC 6600 —la primera supercomputadora que hubo en el mundo— utilizaba este esquema. La CPU Intel 8088 empleada para la IBM PC original utilizaba una versión más débil de este esquema: registros de base, pero sin registros de límite. A partir de la 286, se adoptó un mejor esquema.

<http://laurel.datsi.fi.upm.es/docencia/asignaturas/sod>

Andrew S. Tanenbaum: "Sistemas Operativos Distribuidos", Prentice Hall (1996).

4.2. Intercambio

En un sistema por lotes, la organización de la memoria en particiones fijas es sencilla y efectiva. Cada trabajo se carga en una partición cuando llega al frente de la cola, y permanece en la memoria hasta terminar. En tanto sea posible mantener en la memoria suficientes trabajos para mantener ocupada a la CPU todo el tiempo, no hay por qué usar algo más complicado.

En los sistemas de tiempo compartido o las computadoras personales orientadas a gráficos, la situación es diferente. A veces no hay bastante memoria principal para contener todos los procesos que están activos actualmente, y los procesos en exceso deben mantenerse en disco y traerse dinámicamente para que se ejecuten.

Podemos usar dos enfoques de administración de memoria generales, dependiendo (en parte) del hardware disponible. La estrategia más sencilla, llamada intercambio, consiste en traer a la memoria cada proceso en su totalidad, ejecutarlo durante un tiempo, y después colocarlo otra vez en el disco. La otra estrategia, llamada memoria virtual, permite a los programas ejecutarse aunque sólo estén parcialmente en la memoria principal. A continuación estudiaremos el intercambio; en la sección 4.3 examinaremos la memoria virtual.

El funcionamiento de un sistema con intercambio se ilustra en la Fig. 4-3. Inicialmente, sólo el proceso A está en la memoria. Luego se crean o se traen del disco los procesos B y C. En la Fig. 4-3(d) A termina o se intercambia al disco. Luego llega D y B sale. Por último, entra E.

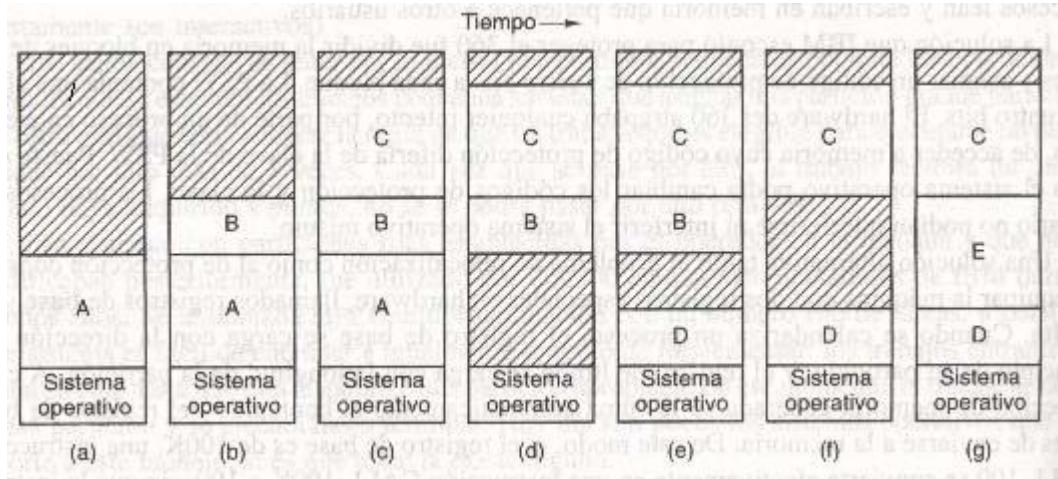


Figura 4-3. La asignación de memoria cambia conforme los procesos entran en la memoria y salen de ella. Las regiones sombreadas son memoria no utilizada.

La diferencia principal entre las particiones fijas de la Fig. 4-2 y las particiones variables de la Fig. 4-3 es que el número, ubicación y tamaño de las particiones varían dinámicamente en el segundo caso conforme los procesos vienen y van, mientras que en el primer caso están fijas. La flexibilidad de no estar atado a un número fijo de particiones que podrían ser demasiado grandes o demasiado pequeñas mejora el aprovechamiento de la memoria, pero también complica la asignación y liberación de memoria, así como su contabilización.

Si el intercambio crea múltiples agujeros en la memoria, es posible combinarlos todos para formar uno grande desplazando todos los procesos hacia abajo hasta donde sea posible. Esta técnica se conoce como compactación de memoria, y pocas veces se practica porque requiere mucho tiempo de CPU. Por ejemplo, en una máquina de 32MB que puede copiar 16 bytes por microsegundo tomaría 2 segundos compactar toda la memoria.

Un punto que cabe destacar se refiere a la cantidad de memoria que debe asignarse a un proceso cuando se crea o se trae a la memoria. Si los procesos se crean con un tamaño fijo que nunca cambia, la asignación es sencilla: se asigna exactamente lo que se necesita, ni más ni menos. En cambio, si los segmentos de datos de los procesos pueden crecer, por ejemplo, mediante asignación dinámica de memoria desde un montículo, como en muchos lenguajes de programación, ocurrirá un problema cada vez que un proceso trate de crecer. Si hay un agujero adyacente al proceso, se le puede asignar y el proceso podrá crecer hacia el agujero. Por otro lado,

si el proceso está adyacente a otro proceso, el proceso en crecimiento tendrá que ser transferido a un agujero en la memoria que tenga el tamaño suficiente para contenerlo, o bien uno o más procesos tendrán que ser intercambiados a disco para crear un agujero del tamaño necesario. Si un proceso no puede crecer en la memoria, y el área de intercambio en el disco está llena, el proceso tendrá que esperar o morir.

Si se espera que la mayor parte de los procesos crezcan durante su ejecución, probablemente será conveniente asignar un poco de memoria adicional cada vez que se traiga un proceso a la memoria o se le cambie de lugar, a fin de reducir el gasto extra asociado al traslado o intercambio de procesos que ya no caben en la memoria que se les asignó. Sin embargo, al intercambiar los procesos al disco, sólo debe intercambiarse la memoria que se está utilizando realmente; sería un desperdicio intercambiar también la memoria adicional. En la Fig. 4-4(a) vemos una configuración de memoria en la que se asignó a dos

Procesos espacio para crecer.

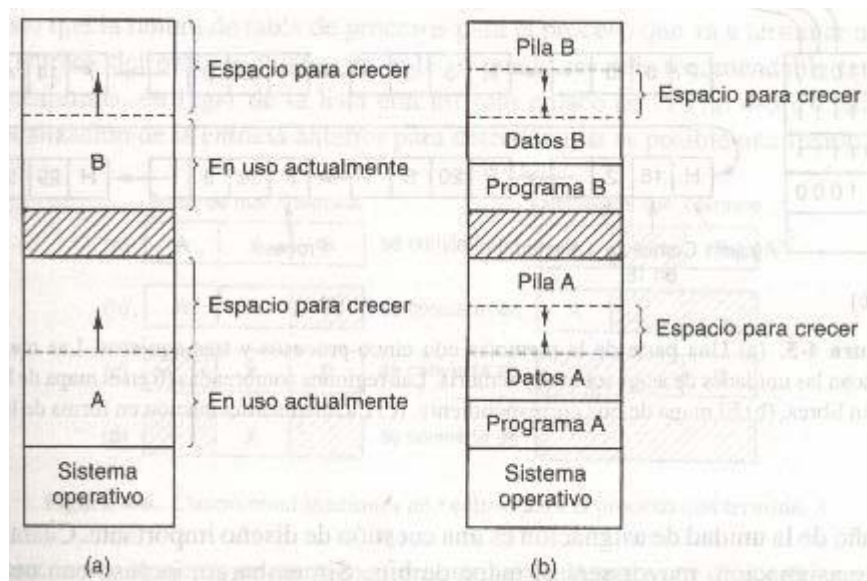


Figura 4-4. (a) Asignación de espacio para un segmento de datos que crece, (b) Asignación de espacio para una pila que crece y un segmento de datos que crece.

Si los procesos pueden tener dos procesos en crecimiento, por ejemplo, el segmento de datos que se está usando como montículo para variables que se asignan y liberan dinámicamente, y un segmento de pila para las variables locales normales y las direcciones de retomo, podemos tener una organización alternativa, que se ilustra en la Fig. 4-4(b). En esta figura vemos que cada proceso tiene una pila en la parte superior de su memoria asignada que está creciendo hacia abajo, y un segmento de datos justo después del texto del programa, creciendo hacia arriba. La memoria que

está entre los dos segmentos se puede destinar a cualquiera de ellos. Si el espacio se agota, el proceso tendrá que ser transferido a un agujero con suficiente espacio, intercambiarse a disco hasta que pueda crearse un agujero del tamaño suficiente, o terminarse.

4.2.1. Administración de memoria con mapas de bits

Cuando la memoria se asigna dinámicamente, el sistema operativo debe administrarla. En términos generales, hay dos formas de contabilizar la utilización de memoria: mapas de bits y listas libres. En esta sección y en la siguiente examinaremos estos dos métodos por turno. Con un mapa de bits, la memoria se divide en unidades de asignación, tal vez sólo de unas cuantas palabras o quizá de varios kilobytes. A cada unidad de asignación corresponde un bit del mapa de bits, que es 0 si la unidad está libre y 1 si está ocupada (o viceversa). En la Fig. 4-5 se muestra una parte de la memoria y el mapa de bits correspondiente.

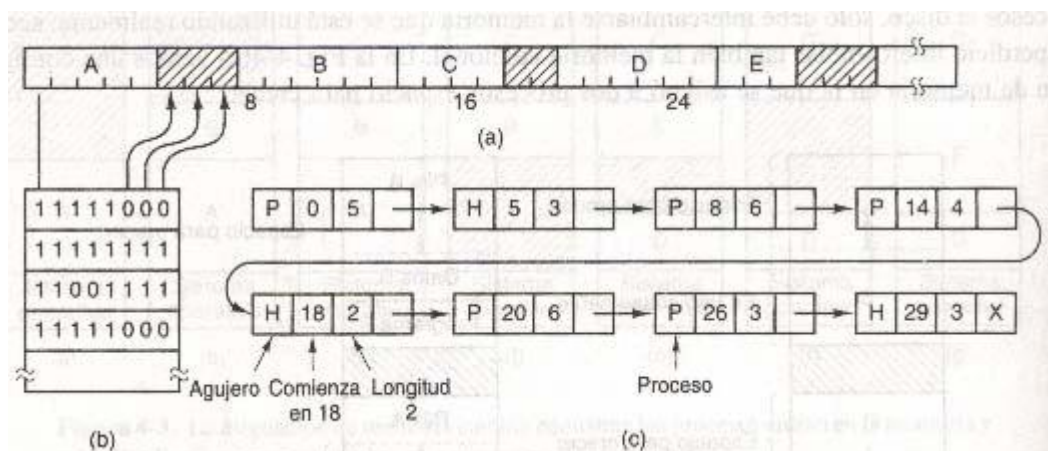


Figura 4-5. (a) Una parte de la memoria con cinco procesos y tres agujeros. Las marcas indican las unidades de asignación de memoria. Las regiones sombreadas (0 en el mapa de bits) están libres, (b) El mapa de bits correspondiente, (c) La misma información en forma de lista.

El tamaño de la unidad de asignación es una cuestión de diseño importante. Cuanto menor sea la unidad de asignación, mayor será el mapa de bits. Sin embargo, incluso con una unidad de asignación de sólo cuatro bits, 32 bits de memoria sólo requerirán un bit del mapa. Una memoria de 32n bits usará n bits de mapa, y el mapa sólo ocupará 1/33 de la memoria. Si se escoge una unidad de asignación grande, el mapa de bits será más pequeño, pero podría desperdiciarse una cantidad apreciable de memoria en la última unidad si el tamaño del proceso no es un múltiplo exacto de la unidad de asignación.

Un mapa de bits ofrece un método sencillo para contabilizar las palabras en una cantidad fija de memoria, porque el tamaño del mapa de bits depende sólo del tamaño de la memoria y del

tamaño de la unidad de asignación. El problema principal que presenta es que una vez que se ha decidido traer a la memoria un proceso de k unidades, el administrador de memoria debe buscar en el mapa de bits una serie de k bits en O consecutivos. La búsqueda de series de una longitud dada en un mapa de bits es una operación lenta (porque la serie puede cruzar fronteras de palabra en el mapa); éste es un argumento en contra de los mapas de bits.

4.2.1.1 Administración de memoria con listas enlazadas

Otra forma de contabilizar la memoria es mantener una lista enlazada de segmentos de memoria libres y asignados, donde un segmento es un proceso o bien un agujero entre dos procesos. La memoria de la Fig. 4-5(a) se representa en la Fig. 4-5(c) como lista enlazada de segmentos. Cada entrada de la lista especifica un agujero (H) o un proceso (P), la dirección en la que principia, la longitud y un apuntador a la siguiente entrada.

En este ejemplo, la lista de segmentos se mantiene ordenada por dirección. Este ordenamiento tiene la ventaja de que cuando un proceso termina o es intercambiado a disco, es fácil actualizar la lista. Un proceso que termina normalmente tiene dos vecinos (excepto cuando está en el tope o la base de la memoria). Éstos pueden ser procesos o agujeros, dando lugar a las cuatro combinaciones de la Fig. 4-6. En la Fig. 4-6(a) la actualización de la lista requiere la sustitución de una P por una H. En las Figs. 4-6(b) y 4-6(c) dos entradas se funden para dar una sola, y el tamaño de la lista se reduce en una entrada. En la Fig. 4-6(d) tres entradas se funden y dos elementos se eliminan de la lista. Puesto que la ranura de tabla de procesos para el proceso que va a terminar normalmente apunta a la entrada del proceso mismo en la lista, puede ser más recomendable tener una lista doblemente enlazada, en lugar de la lista con un solo enlace de la Fig. 4-5(c). Esta Estructura facilita la localización de la entrada anterior para determinar si es posible una fusión.

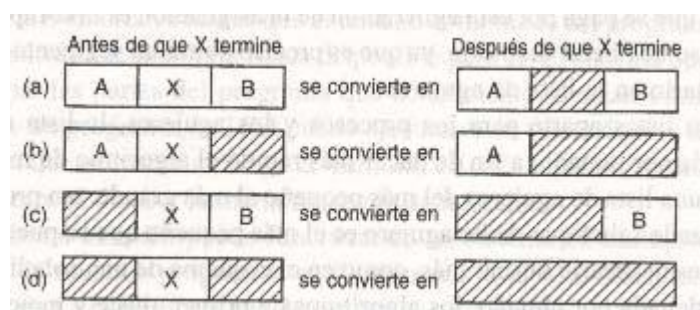


Figura 4-6. Cuatro combinaciones de vecinos para el proceso que termina, X.

Si los procesos y agujeros se mantienen en una lista ordenada por dirección, se pueden usar varios algoritmos para asignar memoria a un proceso recién creado o traído a la memoria. Suponemos que el administrador de memoria sabe cuánta memoria debe asignar. El algoritmo más sencillo es el de primer ajuste. El administrador de memoria examina la lista de segmentos hasta encontrar

un agujero con el tamaño suficiente. A continuación, el agujero se divide en dos fragmentos, uno para el proceso y otro para la memoria desocupada, excepto en el poco probable caso de que el ajuste sea exacto. El algoritmo de primer ajuste es rápido porque la búsqueda es la más corta posible.

Una variante menor del primer ajuste es el siguiente ajuste. Este algoritmo funciona igual que el de primer ajuste, excepto que toma nota de dónde está cada vez que encuentra un agujero apropiado. La siguiente vez que se invoque, el algoritmo comenzará a buscar en la lista a partir del lugar donde se quedó la última vez, en lugar de comenzar por el principio, como hace el primer ajuste. Simulaciones ejecutadas por Bays (1977) demuestran que el siguiente ajuste ofrece un rendimiento ligeramente peor que el primer ajuste.

Otro algoritmo bien conocido es el de mejor ajuste, que examina toda la lista y toma el agujero más pequeño que es adecuado. En lugar de partir un agujero grande que podría necesitarse después, el mejor ajuste trata de encontrar un agujero con un tamaño cercano al que se necesita.

Como ejemplo de primer ajuste y mejor ajuste, consideremos otra vez la Fig. 4-5. Si se requiere un bloque de tamaño 2, el primer ajuste asignará el agujero que está en 5, pero el mejor ajuste asignará el agujero que está en 18. El mejor ajuste es más lento que el primer ajuste porque debe examinar toda la lista cada vez que se invoca. Lo que resulta sorprendente es que también desperdicia más memoria que el primer ajuste o el siguiente ajuste porque tiende a llenar la memoria de pequeños agujeros inútiles. En promedio, el primer ajuste genera agujeros más grandes.

A fin de sortear el problema de partir un agujero con un tamaño casi igual al requerido para obtener un área asignada al proceso y un agujero diminuto, podríamos considerar el peor ajuste, es decir, tomar siempre el agujero más grande disponible, de modo que el agujero sobrante tenga un tamaño suficiente para ser útil. Las simulaciones han demostrado que el peor ajuste tampoco es una idea muy buena.

Los cuatro algoritmos pueden agilizarse manteniendo listas separadas de procesos y agujeros. De este modo, los algoritmos dedican toda su energía a inspeccionar agujeros, no procesos. El inevitable precio que se paga por esta agilización de la asignación es la complejidad adicional y la lentitud del proceso de liberar memoria, ya que es preciso quitar un segmento liberado de la lista de procesos e insertarlo en la lista de agujeros.

Si se mantienen listas aparte para los procesos y los agujeros, la lista de agujeros puede mantenerse ordenada por tamaño, a fin de hacer más rápido el algoritmo de mejor ajuste. Si este algoritmo examina una lista de agujeros del más pequeño al más grande, tan pronto como encuentre un agujero adecuado sabrá que dicho agujero es el más pequeño que se puede usar, y por tanto el mejor ajuste. No es necesario buscar más, como en el esquema de una sola lista. Si se

tiene una lista de agujeros ordenada por tamaño, los algoritmos de primer ajuste y mejor ajuste son igualmente rápidos, y el de siguiente ajuste no

Tiene caso.

Si los agujeros se mantienen en listas aparte de las de procesos, es posible una pequeña optimización. En lugar de tener un conjunto aparte de estructuras de datos para mantener la lista de agujeros, como se hace en la Fig. 4-5(c), se pueden usar los agujeros mismos. La primera palabra de cada agujero podría ser el tamaño del agujero, y la segunda, un apuntador a la siguiente entrada. Los nodos de la lista de la Fig. 4-5(c), que requieren tres palabras y un bit (P/H), ya no son necesarios.

Otro algoritmo de asignación más es el de ajuste rápido, que mantiene listas por separar para algunos de los tamaños más comunes solicitados. Por ejemplo, se podría tener una tabla con n entrada, en la que la primera entrada es un apuntador a una lista de agujeros de 4 K, la segunda entrada es un apuntador a una lista de agujeros de 8K, la tercera es un apuntador a una lista de agujeros de 12K, y así sucesivamente. Los agujeros de, digamos, 21K, podrían colocarse en la lista de 20K o en una lista especial de agujeros con tamaños poco comunes. Si se usa el ajuste rápido, la localización de un agujero del tamaño requerido es extremadamente rápida, pero se tiene la misma desventaja que tienen todos los esquemas que ordenan por tamaño de agujero, a saber, que cuando un proceso termina o es intercambiado a disco la localización de sus vecinos determinar si es posible una fusión es costosa. Si no se efectúan fusiones, la memoria pronto sefragmentará en un gran número de agujeros pequeños en los que no cabrá ningún proceso.

<http://laurel.datsi.fi.upm.es/docencia/asignaturas/sod>

Andrew S. Tanenbaum: "Sistemas Operativos Distribuidos", Prentice Hall (1996).

4.3. Memoria virtual

Hace muchos años las personas enfrentaron por primera vez programas que eran demasiado grandes para caber en la memoria disponible. La solución que normalmente se adoptaba era dividir el programa en fragmentos, llamados superposiciones. La superposición 0 era la primera que se ejecutaba. Al terminar, esta superposición llamaba a otra. Algunos sistemas de superposición eran muy complejos, pues permitían varias superposiciones en la memoria a la vez. Las superposiciones se mantenían en disco y el sistema operativo las intercambiaba con la memoria dinámicamente, según fuera necesario.

Aunque el trabajo real de intercambiar las superposiciones corría por cuenta del sistema, la tarea de dividir el programa en fragmentos tenía que ser efectuada por el programador. La división de programas grandes en fragmentos modulares pequeños consumía tiempo y era tediosa. No pasó mucho tiempo antes de que a alguien se le ocurriera una forma de dejar todo el trabajo a la computadora.

El método que se inventó (Fotheringham, 1961) se conoce ahora como memoria virtual. La idea en que se basa la memoria virtual es que el tamaño combinado del programa, los datos y la pila puede exceder la cantidad de memoria física disponible para él. El sistema operativo mantiene en la memoria principal las partes del programa que actualmente se están usando, y el resto en el disco. Por ejemplo, un programa de 16M puede ejecutarse en una máquina de 4M si se escogen con cuidado los 4M que se mantendrán en la memoria en cada instante, intercambiando segmentos del programa entre el disco y la memoria según se necesite. La memoria virtual también puede funcionar en un sistema de multiprogramación, manteniendo segmentos de muchos programas en la memoria a la vez. Mientras un programa está esperando que se traiga a la memoria una de sus partes, está esperando E/S y no puede ejecutarse, así que puede otorgarse la CPU a otro proceso, lo mismo que en cualquier otro sistema de multiprogramación.

4.3.1. Paginación

La mayor parte de los sistemas de memoria virtual emplean una técnica llamada paginación, que describiremos a continuación. En cualquier computadora, existe un conjunto de direcciones de memoria que los programas pueden producir. Cuando un programa ejecuta una instrucción como MOVE REG, 1000 está copiando el contenido de la dirección de memoria 1000 en REG (o viceversa, dependiendo de la computadora). Las direcciones pueden generarse usando indización, registros de base, registros de segmento y otras técnicas.

Estas direcciones generadas por programas se denominan direcciones virtuales y constituyen el espacio de direcciones virtual. En las computadoras sin memoria virtual, la dirección virtual se coloca directamente en el bus de memoria y hace que se lea o escriba la palabra de memoria física que tiene la misma dirección. Cuando se usa memoria virtual, las direcciones virtuales no pasan directamente al bus de memoria; en vez de ello, se envían a una unidad de administración de memoria (MMU), un chip o colección de chips que transforma las direcciones virtuales en direcciones de memoria física como se ilustra en la Fig. 4-7.

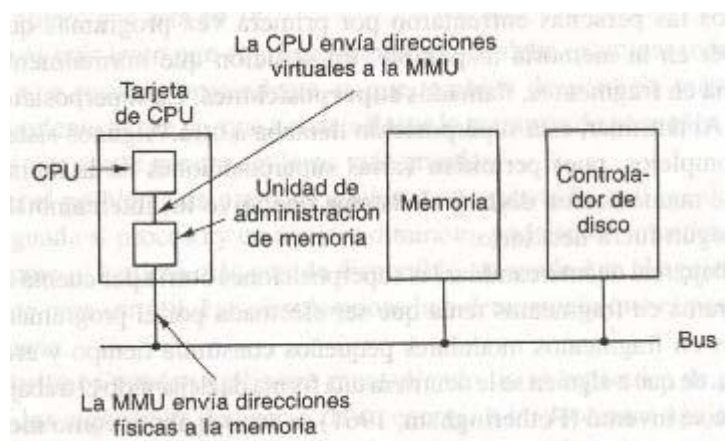


Figura 4-7. La posición y función de la MMU.

En la Fig. 4-8 se muestra un ejemplo muy sencillo de cómo funciona esta transformación. En este ejemplo, tenemos una computadora que puede generar direcciones de 16 bits, desde 0 hasta 64K. Éstas son las direcciones virtuales. Esta computadora, empero, sólo tiene 32K de memoria física, así que si bien es posible escribir programas de 64K, no pueden cargarse enteros en la memoria y ejecutarse. Sin embargo, debe estar presente en el disco una copia completa de la imagen de núcleo del programa, que puede ser de hasta 64K, para poder traer a la memoria fragmento de ella según sea necesario.

El espacio de direcciones virtual se divide en unidades llamadas páginas. Las unidades correspondientes en la memoria física se denominan marcos de página. Las páginas y los marcos de página siempre tienen exactamente el mismo tamaño. En este ejemplo, ese tamaño es 4K, pero es común usar tamaños de página desde 512 bytes hasta 64K en los sistemas existentes. Con 64K de espacio de direcciones virtual y 32K de memoria física, tenemos 16 páginas virtuales y ocho marcos de página. Las transferencias entre la memoria y el disco siempre se efectúan en unidades de una página.

Cuando el programa trata de acceder a la dirección 0, por ejemplo, usando la instrucción `MOVE REG,0` la dirección virtual 0 se envía a la MMU. La MMU ve que esta dirección virtual queda en la

página 0 (0 a 4095) que, según su transformación, es el marco de página 2 (8192 a 12287). Por tanto, la MMU transforma la dirección a 8192 y la coloca en el bus. La tarjeta de memoria nada sabe de la MMU y simplemente ve una solicitud de leer o escribir en la dirección 8192, lo cual hace.

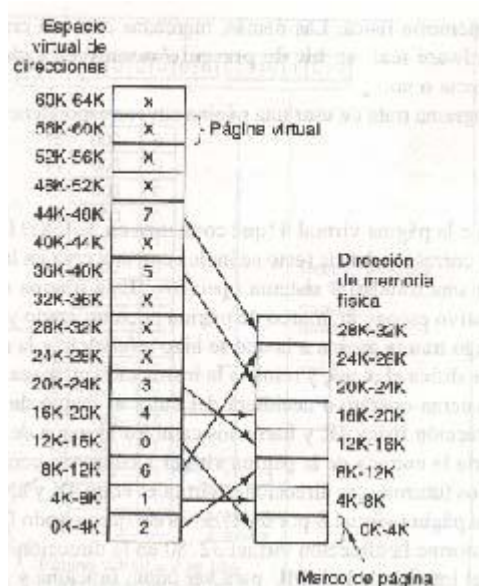


Figura 4-8. La relación entre las direcciones virtuales y las direcciones de la memoria física está dada por la tabla de páginas.

Así, la MMU ha transformado efectivamente todas las direcciones virtuales entre 0 y 4095 en las direcciones físicas 8192 a 12287.

De forma similar, una instrucción `MOVEREG,8192` se transforma efectivamente en `MOVE REG,24576` porque la dirección virtual 8192 está en la página virtual 2 y esta página corresponde al marco de página 6 (direcciones físicas 24576 a 28671). Como tercer ejemplo, la dirección virtual 20500 está 20 bytes después del inicio de la página virtual 5 (direcciones virtuales 20480 a 24575) y se transforma en la dirección física $12288 + 20 = 12308$.

En sí, esta capacidad para transformar las 16 páginas virtuales en cualquiera de los ocho marcos de página ajustando el mapa de la MMU de la forma apropiada no resuelve el problema de que el espacio de direcciones virtual sea más grande que la memoria física. Puesto que sólo tenemos ocho marcos de página físicos, sólo ocho de las páginas virtuales de la Fig. 4-8 tendrán correspondencia con la memoria física. Las demás, marcadas con una cruz en la figura, no se transformarán. En el hardware real, un bit de presente/ausente en cada entrada indica si la página tiene correspondencia o no.

¿Qué sucede si el programa trata de usar una página sin correspondencia, por ejemplo, usando la instrucción

MOVE REG, 32780

Que es el byte 12 dentro de la página virtual 8 (que comienza en 32768)? La MMU se da cuenta de que la página no tiene correspondencia (esto se indica con una cruz en la figura) y hace que la CPU pase el control por una trampa al sistema operativo. Esta trampa se denomina falla de página. El sistema operativo escoge un marco de página poco utilizado y escribe su contenido de vuelta en el disco; luego trae la página a la que se hizo referencia y la coloca en el marco de página recién liberado, modifica el mapa, y reinicia la instrucción atrapada.

Por ejemplo, si el sistema operativo decidiera desalojar el marco de página 1, cargaría la página virtual 8 en la dirección física 4K y haría dos cambios al mapa de la MMU. Primero, el sistema operativo marcaría la entrada de la página virtual 1 como sin correspondencia, a fin de atrapar cualesquier accesos futuros a las direcciones virtuales entre 4K y 8K; luego, reemplazaría la cruz de la entrada de la página virtual 8 por un 1, de modo que cuando la instrucción atrapada se vuelva a ejecutar transforme la dirección virtual 32780 en la dirección física 4108.

Examinemos ahora el interior de la MMU para ver cómo funciona y por qué escogimos un tamaño de página que es una potencia de 2. En la Fig. 4-9 vemos un ejemplo de dirección virtual, 8196 (0010000000000100 en binario), que se transforma usando el mapa de MMU de la Fig. 4-8. La dirección virtual de 16 bits entrante se divide en un número de página de cuatro bits y una distancia de 12 bits. Con cuatro bits para el número de página podemos representar 16 páginas, y con 12 bits para la distancia podemos direccionar los 4096 bytes contenidos en una página.

El número de página se utiliza como índice de la tabla de páginas, produciendo el número del marco de página que corresponde a esa página virtual. Si el bit Presente/ausente es 0, se genera una trampa al sistema operativo. Si el bit es 1, el número de marco de página que se encuentra en la tabla de páginas se copia en los tres bits de orden alto del registro de salida, junto con la distancia de 12 bits, que se copia sin modificación de la dirección virtual entrante. Juntas, estas dos partes forman una dirección física de 15 bits. A continuación el registro de salida se coloca en el bus de memoria como dirección de Memoria física.

4.3.1.1 Tablas de páginas

En teoría, la transformación de direcciones virtuales en físicas se efectúa tal como lo hemos descrito. La dirección virtual se divide en un número de página virtual (bits de orden alto) y una distancia (bits de orden bajo). El número de página virtual sirve como índice para consultar la tabla de páginas y encontrar la entrada correspondiente a esa página virtual. En esa entrada se encuentra el número de marco de página, si lo hay, y este número se anexa al extremo de orden alto de la distancia, sustituyendo al número de página virtual y formando una dirección física que se puede enviar a la memoria.

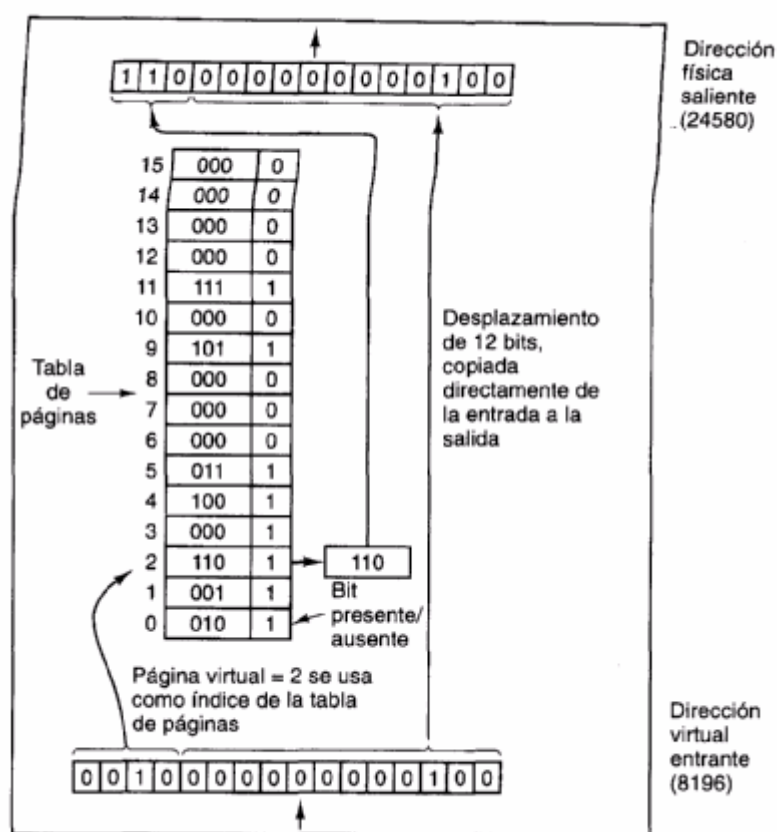


Figura 4-9. Funcionamiento interno de la MMU con 16 páginas de 4K.

El propósito de la tabla de páginas es transformar páginas virtuales en marcos de página. En términos matemáticos, la tabla de página es una función, con el número de página virtual como argumento y el número de marco físico como resultado. Usando el resultado de esta función, el campo de página virtual de una dirección virtual se puede sustituir por un campo de marco de página, formando así una dirección de memoria física.

A pesar de lo sencillo de esta descripción, hay que resolver dos problemas importantes:

1. La tabla de páginas puede ser extremadamente grande.
2. La transformación debe ser rápida.

El primer punto se sigue del hecho de que las computadoras modernas utilizan direcciones virtuales de por lo menos 32 bits. Con un tamaño de página de, digamos, 4K, un espacio de direcciones de 32 bits tiene un millón de páginas, y un espacio de direcciones de 64 bits tiene más de las que nos podamos imaginar. Con un millón de páginas en el espacio de direcciones virtual, la tabla páginas debe tener un millón de entradas. Y recordemos que cada proceso necesita su propia tabla de páginas.

El segundo punto es consecuencia del hecho de que la transformación de virtual a física se debe efectuar en cada referencia a la memoria. Una instrucción típica tiene una palabra de instrucción, y con frecuencia también un operando en memoria. Por tanto, es necesario efectuar una, dos y a veces más referencias a la tabla de página por cada instrucción. Si una instrucción tarda, digamos, 10 ns, la búsqueda en la tabla de páginas debe efectuarse en unos cuantos nanosegundos para evitar que se convierta en un cuello de botella importante.

La necesidad de transformar páginas grandes rápidamente es una restricción significativa sobre la forma de construir computadoras. Aunque el problema es más grave en el caso de las máquinas de mayor capacidad, también es importante en el extremo inferior, donde el costo y la relación precio/rendimiento son críticos. En esta sección y en las siguientes examinaremos el diseño de tablas de páginas con detalle y presentaremos varias soluciones de hardware que se han empleado en computadoras reales.

El diseño más sencillo (al menos en lo conceptual) es tener una sola tabla de páginas que consiste en un arreglo de registros en hardware rápidos, con una entrada para página virtual, indizado por número de página virtual. Cuando se inicia un proceso, el sistema operativo carga los registros con la tabla de páginas del proceso, tomada de una copia que se mantiene en la memoria principal. Durante la ejecución del proceso, no se requieren más referencias a la memoria para la tabla de páginas. Las ventajas de este método es que es sencillo y no requiere referencias a la memoria durante la transformación. Una desventaja es que el costo puede ser alto (si la tabla de páginas es grande). Tener que cargar la tabla de páginas en cada conmutación de contexto también puede perjudicar el rendimiento.

En el otro extremo, la tabla de páginas puede estar toda en la memoria principal. En tal caso, todo lo que el hardware necesita es un solo registro que apunte al principio de la tabla. Este diseño permite modificar el mapa de memoria en una conmutación de contexto volviendo a cargar un registro. Desde luego, la desventaja es que se requiere una o más referencias a la memoria para leer las entradas de la tabla de página durante la ejecución de cada instrucción. Por esta razón

este enfoque casi nunca se usa en su forma más pura, pero a continuación estudiaremos algunas variaciones que tienen un rendimiento muy superior.

4.3.2. Algoritmos De Sustitución De Páginas

Cuando ocurre una falla de página, el sistema operativo tiene que escoger la página que sacará de la memoria para que pueda entrar la nueva página. Si la página que se eliminará fue modificada mientras estaba en la memoria, se debe reescribir en el disco a fin de actualizar la copia del disco, pero si no fue así (p. ej., si la página contenía texto de programa), la copia en disco ya estará actualizada y no será necesario reescribirla. La nueva página simplemente sobrescribe la que está siendo desalojada.

Si bien sería posible escoger una página al azar para ser desalojada cuando ocurre una falla de página, el rendimiento del sistema es mucho mejor si se escoge una página que no se usa mucho. Si se elimina una página de mucho uso, probablemente tendrá que traerse pronto a la memoria otra vez, aumentando el gasto extra. Se ha trabajado mucho sobre el tema de los algoritmos de reemplazo de páginas, tanto teórica como experimentalmente. A continuación describimos algunos de los algoritmos más importantes.

4.3.2.1 El algoritmo de sustitución de páginas óptimo

El mejor algoritmo de reemplazo de páginas posible es fácil de describir pero imposible de implementar. En el momento en que ocurre una falla de páginas, algún conjunto de páginas está en la memoria. A una de estas páginas se hará referencia en la siguiente instrucción (la página que contiene esa instrucción). Otras páginas podrían no necesitarse sino hasta 10, 100 o tal vez 1000 instrucciones después. Cada página puede rotularse con el número de instrucciones que se ejecutarán antes de que se haga referencia a esa página.

El algoritmo de reemplazo de páginas óptimo simplemente dice que se debe eliminar la página que tenga el rótulo más alto. Si una página no se va a usar sino hasta después de 8 millones de instrucciones y otra página no se usará sino hasta después de 6 millones de instrucciones, el desalojo de la primera postergará la falla de página que la traerá de nuevo a la memoria lo más lejos hacia el futuro que es posible. Las computadoras, al igual que las personas, tratan de aplazar los sucesos desagradables el mayor tiempo que se puede.

El único problema con este algoritmo es que no se puede poner en práctica. En el momento en que ocurre la falla de página, el sistema operativo no tiene manera de saber cuándo se hará referencia a cada una de las páginas. (Vimos una situación similar antes con el algoritmo de planificación del primer trabajo más corto; ¿cómo puede el sistema saber cuál trabajo es el más

corto?) No obstante, si se ejecuta un programa en un simulador y se toma nota de todas las referencias a páginas, es posible implementar el reemplazo de páginas óptimo en la segunda ejecución utilizando la información recabada durante la primera.

De este modo, es posible comparar el rendimiento de los algoritmos realizables con el mejor posible. Si un sistema operativo logra un rendimiento, digamos, sólo 1 % peor que el algoritmo óptimo, los esfuerzos dedicados a buscar un mejor algoritmo redundarán cuando más en una mejora del 1%. Para evitar confusiones, debemos dejar sentado que esta bitácora de referencias a páginas sólo es válida para el programa que acaba de medirse. El algoritmo de reemplazo de páginas derivado de él es específico para éste y sólo ese programa. Aunque este método es útil para evaluar los algoritmos de reemplazo de páginas, no sirve de nada en los sistemas prácticos. A continuación estudiaremos algoritmos que sí son útiles en los sistemas reales.

4.3.2.2 El algoritmo de sustitución de páginas no usadas recientemente

Para que el sistema operativo pueda recabar datos estadísticos útiles sobre cuáles páginas se están usando y cuáles no, la mayor parte de las computadoras con memoria virtual tienen dos bits de situación asociados a cada página. R se enciende cada vez que se hace referencia a la página (lectura o escritura). M se enciende cuando se escribe la página (es decir, se modifica). Los bits están contenidos en cada entrada de la tabla de páginas, como se muestra en la Fig. 4-11. Es importante darse cuenta de que estos bits se deben actualizar en cada referencia a la memoria, así que es vital que sea el hardware quien los ajuste. Una vez puesto en 1 un bit, seguirá siendo 1 hasta que el sistema operativo lo ponga en 0 en software.

Si el hardware no tiene estos bits, pueden simularse como sigue. Cuando se inicia un proceso, todas sus entradas de la tabla de páginas se marcan como que no están en la memoria. Tan pronto como se haga referencia a cualquier página, ocurrirá una falla de página. Entonces, el sistema operativo enciende el bit R (en sus tablas internas), modifica la entrada de la tabla de páginas de modo que apunte a la página correcta, con el modo SÓLO LECTURA, y reinicia la instrucción. Si subsecuentemente se escribe en la página, ocurrirá otra falla de página, lo que permitirá el sistema operativo encender el bit M y cambiar el modo de la página a LECTURA/ESCRITURA.

Los bits R y M pueden servir para construir un algoritmo de paginación sencillo como sigue. Cuando se inicia un proceso, el sistema operativo pone en 0 los dos bits de todas sus páginas. Periódicamente (p. ej., en cada interrupción de reloj), se apaga el bit R, a fin de distinguir páginas a las que no se ha hecho referencia recientemente de las que sí se han leído.

Cuando ocurre una falla de página, el sistema operativo examina todas las páginas y las divide en cuatro categorías con base en los valores actuales de sus bits R y M:

Clase 0: no referida, no modificada.

Clase 1: no referida, modificada.

Clase 2: referida, no modificada.

Clase 3: referida, modificada.

Aunque a primera vista las páginas de clase 1 parecen imposibles, ocurren cuando una interrupción del reloj apaga el bit R de una página de clase 3. Las interrupciones de reloj no borran el bit M esta información se necesita para determinar si hay que reescribir en disco la página o no.

El algoritmo NRU (no utilizada recientemente) elimina una página al azar de la clase no vacía que tiene el número más bajo. Este algoritmo supone que es mejor eliminar una página modificada a la que no se ha hecho referencia en por lo menos un tic del reloj (por lo regular 20 ms) que una página limpia que no se está usando mucho. El atractivo principal de NRU es que es fácil de entender, eficiente de implementar y tiene un rendimiento que, si bien ciertamente no es óptimo, a menudo es adecuado.

4.3.2.3 El algoritmo de sustitución de páginas de primera que entra, primera que sale (FIFO)

Otro algoritmo de paginación con bajo gasto extra es el algoritmo FIFO (primera que entra, primera que sale). Para ilustrar su funcionamiento, consideremos un supermercado que tiene suficientes anaqueles para exhibir exactamente k productos distintos. Un día, alguna compañía introduce un nuevo alimento: yogurt orgánico liofilizado instantáneo que puede reconstituirse en un horno de microondas. Su éxito es inmediato, así que nuestro supermercado finito tiene que deshacerse de un producto viejo para poder tener el nuevo en exhibición.

Una posibilidad consiste en encontrar el producto que el supermercado ha tenido en exhibición durante más tiempo (es decir, algo que comenzó a vender hace 120 años) y deshacerse de él bajo el supuesto de que a nadie le interesa ya. En efecto, el supermercado mantiene una lista enlazada de todos los productos que vende en el orden en que se introdujeron. El nuevo se coloca al final de la lista; el que está a la cabeza de la lista se elimina.

Como algoritmo de reemplazo de páginas, puede aplicarse la misma idea. El sistema operativo mantiene una lista de todas las páginas que están en la memoria, siendo la página que está a la cabeza de la lista la más vieja, y la del final, la más reciente. Cuando hay una falla de página, se elimina la página que está a la cabeza de la lista y se agrega la nueva página al final. Cuando FIFO se aplica a una tienda, el producto eliminado podría ser cera para el bigote, pero también podría ser harina, sal o mantequilla. Cuando FIFO se aplica a computadoras, surge el mismo problema. Por esta razón, casi nunca se usa FIFO

En su forma pura.

4.3.2.4 El algoritmo de sustitución de páginas de segunda oportunidad

Una modificación sencilla de FIFO que evita el problema de desalojar una página muy utilizada consiste en inspeccionar el bit R de la página más vieja. Si es 0, sabremos que la página, además de ser vieja, no ha sido utilizada recientemente, así que la reemplazamos de inmediato. Si el bit R es 1, se apaga el bit, se coloca la página al final de la lista de páginas, y se actualiza su tiempo de carga como si acabara de ser traída a la memoria. Luego continúa la búsqueda.

El funcionamiento de este algoritmo, llamado de segunda oportunidad, se muestra en la Fig. 4-13. En la Fig. 4-13(a) vemos que se mantienen las páginas de la A a la H en una lista enlazada ordenadas según el momento en que se trajeron a la memoria.

Supongamos que ocurre una falla de página en el tiempo 20. La página más antigua es A, que legó en el tiempo 0, cuando se inició el proceso. Si A tiene el bit R apagado, es desalojada de la memoria, ya sea escribiéndose en el disco (si está sucia) o simplemente abandonándose (si está limpia). En cambio, si el bit R está encendido, A se coloca al final de la lista y su "tiempo de carga" se ajusta al tiempo actual (20). También se apaga su bit R. La búsqueda de una página apropiada continúa con B.

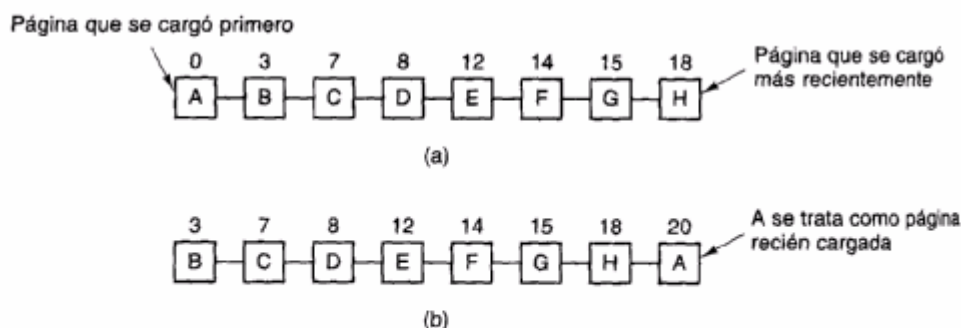


Figura 4-13. Funcionamiento del algoritmo de segunda oportunidad. (a) Páginas ordenadas en orden FIFO. (b) Lista de páginas si ocurre una falla de página en el tiempo 20 y A tiene su bit R encendido.

Lo que hace el algoritmo de segunda oportunidad es buscar una página vieja a la que no se, haya hecho referencia en el intervalo de reloj anterior. Si se ha hecho referencia a todas las páginas, este algoritmo pasa a ser FIFO puro.

Específicamente, imaginemos que todas las páginas de la Fig. 4-13(a) tienen su bit R encendido. Una por una, el sistema operativo pasará páginas al final de la lista, apagando el bit R cada vez que anexa una página al final de la lista. Tarde o temprano, regresará a la página A, que ahora tiene su bit R apagado. En este punto, A será desalojada. Así, el algoritmo siempre termina.

4.3.2.5 El algoritmo de sustitución de páginas por reloj

Aunque el algoritmo de segunda oportunidad es razonable, es innecesariamente eficiente porque constantemente cambia de lugar páginas dentro de su lista. Un enfoque mejor consiste en mantener todas las páginas en una lista circular con forma de reloj, como se muestra en la Fig. 4-14. Una manecilla apunta a la página más vieja.

Cuando ocurre una falla de página, se inspecciona la página a la que apunta la manecilla. Si su bit R es 0, se desaloja la página, se inserta la nueva página en el reloj en su lugar, y la manecilla avanza una posición. Si R es 1, se pone en 0 y la manecilla se avanza a la siguiente página. Este proceso se repite hasta encontrar una página con R = 0. No resulta sorprendente que este algoritmo se llame de reloj. La única diferencia respecto al de segunda oportunidad es la implementación.

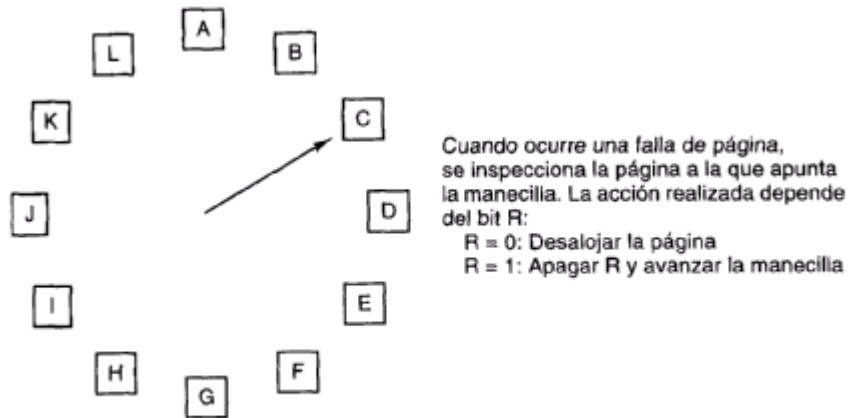


Figura 4-14. Algoritmo de reemplazo de páginas por reloj.

<http://laurel.datsi.fi.upm.es/docencia/asignaturas/sod>

Andrew S. Tanenbaum: "Sistemas Operativos Distribuidos", Prentice Hall (1996).

4.4. Ejercicios por temas

◆ Ejercicio Tema 1.

Administración de la memoria sin intercambio o paginación

Para los siguientes enunciados elija la respuesta correcta:

1. El propósito de la administración de Ram es:
 - a. saber cómo se ejecutan los procesos
 - b. saber cómo se encuentra la ram en el procesamiento
 - c. saber que información envía la ram al disco
 - d. saber cómo se encuentra el disco
2. La fragmentación interna se da por:
 - a. que una partición no es usada
 - b. una partición es usada parcialmente por un proceso
 - c. sobran muchas particiones
 - d. todo el espacio de la ram esta ocupado

◆ Ejercicio Tema 2

Intercambio

Como se realiza un proceso de compactación en los esquemas de intercambio

En los esquemas de crecimiento quien decide sobre la magnitud del crecimiento para cada proceso

Porque son obsoletos los mapas de bits respecto a la representación de listas ligadas.

◆ Ejercicio Tema 3

Memoria virtual

Como es posible que 8 procesos de 1Mb cada uno se puedan ejecutar en una memoria de 1Mb

Explique en que consiste la paginación y donde se aplica

Describe el proceso de un fallo de página

Para que sirven los algoritmos de reemplazo de página

Explique cómo trabaja la segmentación

5. INTRODUCCIÓN A LOS SISTEMAS OPERATIVOS DISTRIBUIDOS

OBJETIVO GENERAL

- ◆ Conocer características generales de los sistemas operativos actuales que permitan facilitar su uso y aplicación

OBJETIVOS ESPECÍFICOS

- ◆ Conocer las condiciones mínimas de una comunicación entre computadoras
- ◆ Definir los recursos de conectividad
- ◆ Conocer los protocolos básicos de comunicación
- ◆ Entender el acoplamiento de procesadores

Prueba Inicial

- ◆ Explicar para qué sirve el modelo OSI en un sistema distribuido
- ◆ Diga como un servidor de red controla el tráfico en un sistemas distribuido
- ◆ Explique los conceptos fuertemente acoplado y débilmente acooles

5.1. Sistemas Distribuidos

<http://www.augcyl.org/?q=glol-intro-sistemas-distribuidos>

George Coulouris, Jean Dollimore, Tim Kindberg: "Distributed Systems, Concepts and Design", Addison-Wesley (1994).]

En el inicio de la era de la informática las computadoras eran grandes y caras. Debido a su escasez y coste, éstas funcionaban de forma independiente entre ellas.

A partir de los años 70, surgen los primeros miniordenadores, que competirían con los grandes ordenadores tanto por las prestaciones como por su precio, con lo que se extendió su uso. Los grandes sistemas centralizados fueron dejando paso lentamente a sistemas mucho más descentralizados, y formados por varios ordenadores o a sistemas multiprocesador. Pronto surgieron nuevas necesidades de interconexión de los equipos, y se desarrollaron las redes de área local (LAN), como Ethernet o Token ring. En la actualidad, Internet es la red de mayor tamaño y la más usada, y mantiene un impresionante ritmo de crecimiento. Además, Internet es la base de muchos nuevos proyectos de sistemas distribuidos.

Aunque los actuales sistemas de red solucionan parte de las necesidades actuales de comunicación entre computadoras, tienen importantes limitaciones, y no son aplicables a una

gran cantidad de problemas. Por ello surge la necesidad de crear sistemas distribuidos que sustituyan a los actuales sistemas de red o a los sistemas multiprocesadores.

◆ **Conceptos de los sistemas distribuidos**

Los sistemas distribuidos están basados en las ideas básicas de transparencia, eficiencia, flexibilidad, escalabilidad y fiabilidad. Sin embargo estos aspectos son en parte contrarios, y por lo tanto los sistemas distribuidos han de cumplir en su diseño el compromiso de que todos los puntos anteriores sean solucionados de manera aceptable.

◆ **Transparencia**

El concepto de transparencia de un sistema distribuido va ligado a la idea de que todo el sistema funcione de forma similar en todos los puntos de la red, independientemente de la posición del usuario. Queda como labor del sistema operativo el establecer los mecanismos que oculten la naturaleza distribuida del sistema y que permitan trabajar a los usuarios como si de un único equipo se tratara.

En un sistema transparente, las diferentes copias de un archivo deben aparecer al usuario como un único archivo. Queda como labor del sistema operativo el controlar las copias, actualizarlas en caso de modificación y en general, la unicidad de los recursos y el control de la concurrencia.

El que el sistema disponga de varios procesadores debe lograr un mayor rendimiento del sistema, pero el sistema operativo debe controlar que tanto los usuarios como los programadores vean el núcleo del sistema distribuido como un único procesador. El paralelismo es otro punto clave que debe controlar el sistema operativo, que debe distribuir las tareas entre los distintos procesadores como en un sistema multiprocesador, pero con la dificultad añadida de que ésta tarea hay que realizarla a través de varios ordenadores.

◆ **Eficiencia**

La idea base de los sistemas distribuidos es la de obtener sistemas mucho más rápidos que los ordenadores actuales. Es en este punto cuando nos encontramos de nuevo con el paralelismo.

Para lograr un sistema eficiente hay que descartar la idea de ejecutar un programa en un único procesador de todo el sistema, y pensar en distribuir las tareas a los procesadores libres más rápidos en cada momento.

La idea de que un procesador vaya a realizar una tarea de forma rápida es bastante compleja, y depende de muchos aspectos concretos, como la propia velocidad del procesador, pero también la localidad del procesador, los datos, los dispositivos, etc. Se han de evitar situaciones como enviar un trabajo de impresión a un ordenador que no tenga conectada una impresora de forma local.

◆ **Flexibilidad**

Un proyecto en desarrollo como el diseño de un sistema operativo distribuido debe estar abierto a cambios y actualizaciones que mejoren el funcionamiento del sistema. Esta necesidad ha provocado una diferenciación entre las dos diferentes arquitecturas del núcleo del sistema operativo: el núcleo monolítico y el micronúcleo. Las diferencias entre ambos son los servicios que ofrece el núcleo del sistema operativo. Mientras el núcleo monolítico ofrece todas las funciones básicas del sistema integradas en el núcleo, el micronúcleo incorpora solamente las fundamentales, que incluyen únicamente el control de los procesos y la comunicación entre ellos y la memoria. El resto de servicios se cargan dinámicamente a partir de servidores en el nivel de usuario.

◆ **Núcleo monolítico**

Como ejemplo de sistema operativo de núcleo monolítico está UNIX. Estos sistemas tienen un núcleo grande y complejo, que engloba todos los servicios del sistema. Está programado de forma no modular, y tiene un rendimiento mayor que un micronúcleo. Sin embargo, cualquier cambio a realizar en cualquier servicio requiere la parada de todo el sistema y la recompilación del núcleo.

◆ **Micronúcleo**

La arquitectura de micronúcleo ofrece la alternativa al núcleo monolítico. Se basa en una programación altamente modular, y tiene un tamaño mucho menor que el núcleo monolítico. Como consecuencia, el refinamiento y el control de errores son más rápidos y sencillos. Además, la actualización de los servicios es más sencilla y ágil, ya que sólo es necesaria la recompilación del servicio y no de todo el núcleo. Como contraprestación, el rendimiento se ve afectado negativamente.

En la actualidad la mayoría de sistemas operativos distribuidos en desarrollo tienden a un diseño de micronúcleo. Los núcleos tienden a contener menos errores y a ser más fáciles de implementar y de corregir. El sistema pierde ligeramente en rendimiento, pero a cambio consigue un gran aumento de la flexibilidad.

◆ **Escalabilidad**

Un sistema operativo distribuido debería funcionar tanto para una docena de ordenadores como varios millares. Igualmente, debería no ser determinante el tipo de red utilizada (LAN o WAN) ni las distancias entre los equipos, etc.

Aunque este punto sería muy deseable, puede que las soluciones válidas para unos cuantos ordenadores no sean aplicables para varios miles. Del mismo modo el tipo de red condiciona tremendamente el rendimiento del sistema, y puede que lo que funcione para un tipo de red, para otro requiera un nuevo diseño.

La escalabilidad propone que cualquier ordenador individual ha de ser capaz de trabajar independientemente como un sistema distribuido, pero también debe poder hacerlo conectado a muchas otras máquinas.

2. Fiabilidad

Una de las ventajas claras que nos ofrece la idea de sistema distribuido es que el funcionamiento de todo el sistema no debe estar ligado a ciertas máquinas de la red, sino que cualquier equipo pueda suplir a otro en caso de que uno se estropee o falle.

La forma más evidente de lograr la fiabilidad de todo el sistema está en la redundancia. La información no debe estar almacenada en un solo servidor de archivos, sino en por lo menos dos máquinas. Mediante la redundancia de los principales archivos o de todos evitamos el caso de que el fallo de un servidor bloquee todo el sistema, al tener una copia idéntica de los archivos en otro equipo.

Otro tipo de redundancia más compleja se refiere a los procesos. Las tareas críticas podrían enviarse a varios procesadores independientes, de forma que el primer procesador realizaría la tarea normalmente, pero ésta pasaría a ejecutarse en otro procesador si el primero hubiera fallado.

Comunicación

La comunicación entre procesos en sistemas con un único procesador se lleva a cabo mediante el uso de memoria compartida entre los procesos. En los sistemas distribuidos, al no haber conexión física entre las distintas memorias de los equipos, la comunicación se realiza mediante la transferencia de mensajes.

3. El estándar ISO OSI

Para el envío de mensajes se usa el estándar ISO OSI (interconexión de sistemas abiertos), un modelo por capas para la comunicación de sistemas abiertos. Las capas proporcionan varias interfaces con diferentes niveles de detalle, siendo la última la más general. El estándar OSI define las siguientes siete capas: física, enlace de datos, red, transporte, sesión, presentación y aplicación.

El modelo OSI distingue dos tipos de protocolos, los orientados hacia las conexiones y los protocolos sin conexión. En los primeros, antes de cualquier envío de datos se requiere una conexión virtual, que tras el envío deben finalizar. Los protocolos sin conexión no requieren este paso previo, y los mensajes se envían en forma de datagramas.

4. Modo de transmisión asíncrona ATM

El modo de transmisión asíncrona o ATM proporciona un rápido modo de transmisión. Las altas velocidades se alcanzan prescindiendo de la información de control de flujo y de control de errores

en los nodos intermedios de la transmisión. ATM usa el modo orientado a conexión, y permite la transmisión de diferentes tipos de información, como voz, vídeo, datos, etc.

El modelo cliente-servidor basa la comunicación en una simplificación del modelo OSI. Las siete capas que proporciona producen un desaprovechamiento de la velocidad de transferencia de la red, con lo que sólo se usarán tres capas: física (1), enlace de datos (2) y solicitud/respuesta (5). Las transferencias se basan en el protocolo solicitud/respuesta y se elimina la necesidad de conexión.

◆ RPC

Otro paso en el diseño de un sistema operativo distribuido plantea las llamadas a procedimientos remotos o RPCs. Los RPC amplían la llamada local a procedimientos, y los generalizan a una llamada a un procedimiento localizado en cualquier lugar de todo el sistema distribuido. En un sistema distribuido no se debería distinguir entre llamadas locales y RPCs, lo que favorece en gran medida la transparencia del sistema.

Una de las dificultades más evidentes a las que se enfrenta el RPC es el formato de los parámetros de los procedimientos. Un ejemplo para ilustrar este problema es la posibilidad de que en un sistema distribuido formado por diferentes tipos de ordenadores, un ordenador con formato little endian llamara a un procedimiento de otro ordenador con formato big endian, etc. Este problema se podría solucionar si tenemos en cuenta que ambos programas conocen el tipo de datos de los parámetros, o estableciendo un estándar en el formato de los parámetros, de forma que sea usado de forma única.

Otro problema de peor solución es el paso de apuntadores como parámetros. Debido a que los apuntadores guardan una dirección del espacio de direcciones local, el procedimiento que recibe el apuntador como parámetro no puede usar inmediatamente el apuntador, ya que no tiene acceso a los datos, que para él son remotos. En el tema 7 se describirá la memoria compartida, que propone una solución a este problema.

Por último queda por solucionar la tolerancia a fallos. Una llamada a un procedimiento remoto puede fallar por motivos que antes no existían, como la pérdida de mensajes o el fallo del cliente o del servidor durante la ejecución del procedimiento.

La limitación del RPC más clara en los sistemas distribuidos es que no permite enviar una solicitud y recibir respuesta de varias fuentes a la vez, sino que la comunicación se realiza únicamente entre dos procesos. Por motivos de tolerancia a fallos, bloqueos, u otros, sería interesante poder tratar la comunicación en grupo.

5. Comunicación en grupo

La comunicación en grupo tiene que permitir la definición de grupos, así como características propias de los grupos, como la distinción entre grupos abiertos o que permiten el acceso y cerrados que lo limitan, o como la distinción del tipo de jerarquía dentro del grupo. Igualmente, los grupos han de tener operaciones relacionadas con su manejo, como la creación o modificación.

◆ Sincronización

La sincronización en sistemas de un único ordenador no requiere ninguna consideración en el diseño del sistema operativo, ya que existe un reloj único que proporciona de forma regular y precisa el tiempo en cada momento. Sin embargo, los sistemas distribuidos tienen un reloj por cada ordenador del sistema, con lo que es fundamental una coordinación entre todos los relojes para mostrar una hora única. Los osciladores de cada ordenador son ligeramente diferentes, y como consecuencia todos los relojes sufren un desfase y deben ser sincronizados continuamente. La sincronización no es trivial, porque se realiza a través de mensajes por la red, cuyo tiempo de envío puede ser variable y depender de muchos factores, como la distancia, la velocidad de transmisión o la propia saturación de la red, etc.

◆ El reloj

La sincronización no tiene por qué ser exacta, y bastará con que sea aproximadamente igual en todos los ordenadores. Hay que tener en cuenta, eso sí, el modo de actualizar la hora de un reloj en particular. Es fundamental no retrasar nunca la hora, aunque el reloj adelante. En vez de eso, hay que ralentizar la actualización del reloj, frenarlo, hasta que alcance la hora aproximadamente. Existen diferentes algoritmos de actualización de la hora, tres de ellos se exponen brevemente a continuación.

◆ Algoritmo de Lamport

Tras el intento de sincronizar todos los relojes, surge la idea de que no es necesario que todos los relojes tengan la misma hora exacta, sino que simplemente mantengan una relación estable de forma que se mantenga la relación de qué suceso ocurrió antes que otro suceso cualquiera.

Este algoritmo se encarga exclusivamente de mantener el orden en que se suceden los procesos. En cada mensaje que se envía a otro ordenador se incluye la hora. Si el receptor del mensaje tiene una hora anterior a la indicada en el mensaje, utiliza la hora recibida incrementada en uno para actualizar su propia hora.

◆ **Algoritmo de Cristian**

Consiste en disponer de un servidor de tiempo, que reciba la hora exacta. El servidor se encarga de enviar a cada ordenador la hora. Cada ordenador de destino sólo tiene que sumarle el tiempo de transporte del mensaje, que se puede calcular de forma aproximada.

◆ **Algoritmo de Berkeley**

La principal desventaja del algoritmo de Cristian es que todo el sistema depende del servidor de tiempo, lo cual no es aceptable en un sistema distribuido fiable.

El algoritmo de Berkeley usa la hora de todos los ordenadores para elaborar una media, que se reenvía para que cada equipo actualice su propia hora ralentizando el reloj o adoptando la nueva hora, según el caso.

6. Otros problemas de sincronización

El reloj es únicamente uno de tantos problemas de sincronización que existen en los sistemas distribuidos. A continuación planteamos otros problemas relacionados con la sincronización.

En el momento de modificar unos datos compartidos, los procesos deben lograr la exclusión mutua que garantice que dos procesos no modifiquen los datos a la vez.

Algunos algoritmos distribuidos requieren que un proceso funcione como coordinador. Es necesario establecer ciertos algoritmos de elección de estos procesos.

Es necesario ocultar las técnicas de sincronización mediante la abstracción de las transacciones atómicas, que permitan a los programadores salvar los detalles de la programación con sincronización.

Soluciones frente a bloqueos son bastante más complejas que en sistemas con un único procesador.

7. Sistema de archivos

A diferencia de los sistemas de archivos clásicos, un sistema de archivos distribuido debe ser descentralizado, transparente y tolerante a fallos.

Transparencia

El problema más importante a resolver es el modo de que todos los ordenadores puedan acceder a todos los archivos del sistema. Para ello es necesario que todos los ordenadores lleven siempre y en todo momento una copia actualizada de la estructura de archivos y directorios. Si esta estructura oculta la localización física de los archivos entonces hemos cumplido el criterio de transparencia.

◆ Fallos del sistema

Que el sistema de archivos sea tolerante a fallos implica que el sistema debe guardar varias copias del mismo archivo en distintos ordenadores para garantizar la disponibilidad en caso de fallo del servidor original. Además, se ha de aplicar un algoritmo que nos permita mantener todas las copias actualizadas de forma consistente, o un método alternativo que sólo nos permita acceder al archivo actualizado, como invalidar el resto de copias cuando en cualquiera de ellas se vaya a realizar una operación de escritura. El uso de memorias cache para agilizar el acceso a los archivos también es recomendable, pero este caso requiere analizar con especial atención la consistencia del sistema.

◆ Modelos de acceso

Debido a la complejidad del acceso a los archivos a través de todo el sistema distribuido, surgen dos modelos para el acceso a los archivos: el modelo carga/descarga, y el modelo de acceso remoto. El primer modelo simplifica el acceso permitiendo únicamente las operaciones de cargar y descargar un archivo. El acceso a cualquier parte del archivo implica solicitar y guardar una copia local del archivo completo, y sólo se puede escribir de forma remota el archivo completo. Este método sería especialmente ineficaz a la hora de realizar pequeñas modificaciones en archivos muy grandes, como podrían ser bases de datos. El modelo de acceso remoto es mucho más complejo, y permite todas las operaciones típicas de un sistema de archivos local.

◆ Memoria compartida distribuida

La memoria compartida distribuida o DSM es una abstracción que se propone como alternativa a la comunicación por mensajes.

◆ Memoria compartida basada en páginas

El esquema de DSM propone un espacio de direcciones de memoria virtual que integre la memoria de todas las computadoras del sistema, y su uso mediante paginación. Las páginas quedan restringidas a estar necesariamente en un único ordenador. Cuando un programa intenta acceder a una posición virtual de memoria, se comprueba si esa página se encuentra de forma local. Si no se encuentra, se provoca un fallo de página, y el sistema operativo solicita la página al resto de computadoras. El sistema funciona de forma análoga al sistema de memoria virtual tradicional, pero en este caso los fallos de página se propagan al resto de ordenadores, hasta que la petición llega al ordenador que tiene la página virtual solicitada en su memoria local. A primera vista este sistema parece más eficiente que el acceso a la memoria virtual en disco, pero en la realidad ha mostrado ser un sistema demasiado lento en ciertas aplicaciones, ya que provoca un tráfico de páginas excesivo.

Una mejora dirigida a mejorar el rendimiento sugiere dividir el espacio de direcciones en una zona local y privada y una zona de memoria compartida, que se usará únicamente por procesos que necesiten compartir datos. Esta abstracción se acerca a la idea de programación mediante la declaración explícita de datos públicos y privados, y minimiza el envío de información, ya que sólo se enviarán los datos que realmente vayan a compartirse.

◆ **Memoria compartida basada en objetos**

Una alternativa al uso de páginas es tomar el objeto como base de la transferencia de memoria. Aunque el control de la memoria resulta más complejo, el resultado es al mismo tiempo modular y flexible, y la sincronización y el acceso se pueden integrar limpiamente. Otra de las restricciones de este modelo es que todos los accesos a los objetos compartidos han de realizarse mediante llamadas a los métodos de los objetos, con lo que no se admiten programas no modulares y se consideran incompatibles.

◆ **Modelos de consistencia**

La duplicidad de los bloques compartidos aumenta el rendimiento, pero produce un problema de consistencia entre las diferentes copias de la página en caso de una escritura. Si con cada escritura es necesario actualizar todas las copias, el envío de las páginas por la red provoca que el tiempo de espera aumente demasiado, convirtiendo este método en impracticable. Para solucionar este problema se proponen diferentes modelos de consistencia, que establezcan un nivel aceptable de acercamiento tanto a la consistencia como al rendimiento. Nombramos algunos modelos de consistencia, del más fuerte al más débil: consistencia estricta, secuencial, causal, PRAM, del procesador, débil, de liberación y de entrada.

<http://www.augcyl.org/?q=glol-intro-sistemas-distribuidos>

George Coulouris, Jean Dollimore, Tim Kindberg: "Distributed Systems, Concepts and Design", Addison-Wesley (1994).]

5.2. Ejercicios por temas

◆ Ejercicio del tema 1

Objetivos y Conceptos de Hardware

1. represente una red distribuida con todas sus partes y diga su funcionalidad.
2. La fibra óptica, el par trenzado y el Utp presentan diferencias que debemos describir. Escriba al menos tres.
¿Cuáles son las características para un servidor principal que se debe tener en una red distribuida?

◆ Ejercicio tema 2

Conceptos de Software

1. Grafique y explique cada una de las capas del modelo OSI
2. ¿Qué es un protocolo de red y cómo se usan?

◆ Ejercicio tema 3

Diseño y Comunicación

1. ¿Cuáles son las características de una red LAN y representarla?
2. ¿Cuáles son las características de una red WAN y representarla?
3. ¿Cómo administramos recursos en un sistema distribuido?

Actividad

Con un ejemplo real de una empresa grande o multifuncional represente el sistema distribuido que la soporta indicando sus características de hardware, software y su diseño e implantación

PISTAS DE APRENDIZAJE (faltan)

Son las ayudas que presenta el experto temático o profesor que elabora el módulo para que el estudiante no olvide detalles del tema que podrían hacer flaquear el proceso de aprendizaje.

Las pistas que coloca aquí, las selecciona el experto temático de acuerdo a su experiencia en el tema como docente presencial, a distancia o virtual. Estas pistas tienen mucho que ver con lo que al estudiante se le olvida de manera frecuente y repetitiva. Coloque la pista y su explicación de acuerdo a los temas del módulo (mínimo 8).

Utilice las palabras clave de las pistas de aprendizaje: Tener en cuenta, Tenga presente o Traer a la memoria.

Ejemplo

Tener en cuenta: Los neurotransmisores son moléculas pequeñas que son enviadas por una neurona a otra para salvar un "espacio vacío"

Tenga presente a la hora de evaluar un paciente cuando el tema de consulta es que está nervioso o fatigado: Sistema nervioso central, Sistema nervioso periférico y Sistema nervioso autónomo o vegetativo

Traer a la memoria Las señales químicas del sistema nervioso: neurotransmisores y hormonas

5.3. Glosario

Lenguaje Máquina: (instrucciones)_son aquellas que basadas en secuencias de ceros y unos (código binario), son entendibles directamente por el ordenador y por lo tanto ejecutables sin necesidad de traducción.

Máquina Virtual: es aquella interface que mantiene una máquina mediante la cual nos comunicamos con los dispositivos hardware del ordenador. De esta forma nosotros trabajamos a un nivel superior eliminando la complejidad de dichos dispositivos. De esta forma podemos mantener distintos Sistemas Operativos corriendo sobre una misma máquina.

Memoria Dinámica: zona de la memoria principal que reservan los compiladores para poder utilizarla mediante una serie de peticiones al gestor de memoria en tiempo de ejecución.

Memoria Principal: dispositivo que almacena información en forma de datos codificados en binario. Es accedida directamente por la CPU y además es de lectura y escritura, pero volátil. Está formada por una serie de posiciones o celdillas de memoria, las cuales son referenciadas mediante un sistema de direcciones lógicas.

Memoria Secundaria: tipo de memoria auxiliar que retiene toda la información almacenada cuando se desconecta de la fuente de alimentación. Es de acceso lento pero de gran capacidad.

Multiprogramación: sistema de programación que permite la ejecución simultanea de varios procesos residentes en la memoria principal. Es decir, todos los procesos avanzan en su ejecución

Multiusuario: sistema informático capaz de soportar el trabajo de varios usuarios en una misma máquina o grupo reducido de ellas.

Periféricos: serán aquellos dispositivos que no forman parte del computador central estando este formado por la memoria principal y la CPU. Sirven para comunicarse con el mismo

Pila: zona reservada de la memoria o registros hardware donde se almacena temporalmente el estado o información de un programa, rutina, etc... . Mantiene una política de inserciones y eliminaciones LIFO (Primero en entrar Último en salir).

Puerto: dispositivo físico que conecta los canales internos de información del ordenador con las líneas de comunicación de los posibles periféricos conectables.

Registro: pequeña memoria interna del microprocesador, formada generalmente por biestables. Es de rápido acceso y son de muy utilizados programando en lenguaje ensamblador.

Rutinas De E/S: conjunto de instrucciones encargado de controlar la transferencia de datos entre los periféricos y el ordenador. Gestionan las capacidades de los puertos.

Serie: método de procesamiento o transmisión de datos basado en etapas sucesivas, no simultáneas. Ejemplo de transmisión bit a bit.

Software: son los programas, incluyendo procedimientos, utilidades, sistemas operativos, programas de aplicación y paquetes informáticos, implementados para un sistema informático.

Tiempo Compartido: sistema de reparto de la capacidad de proceso basado en la división del tiempo de CPU entre los distintos trabajos que hay en memoria principal, para que todos avancen en su ejecución.

Traductor: programa que convierte mediante una traducción un lenguaje fuente en un lenguaje objeto, sin que por ello varíe la semántica del código traducido, ya que tan solo cambia su representación.

Abstracción de Proceso: método utilizado por un Sistema Operativo para representar un proceso tanto de forma interna en memoria, como externamente para mostrar sus características al usuario.

Apuntador o Puntero: tipo de dato formado por una dirección de memoria principal. A través de este tipo de dato se puede acceder a cualquier bloque de memoria que esté referenciado, es decir que sepamos su dirección lógica.

Carga del Sistema: número de procesos (programas activos), que tenemos actualmente cargados en memoria principal. Sobre cualquiera de ellos puede actuar el planificador.

Contexto o Entorno: término que engloba a las principales características de un proceso que nos proporcionan información sobre el entorno del mismo: Contador de Programa, registros del procesador, información de la pila y sus principales atributos.

Cola: estructura de datos dinámica residente en memoria principal derivada de las listas, la cual puede cambiar de tamaño en tiempo de ejecución. Las inserciones y eliminaciones se van a realizar atendiendo al método FIFO (Primero en entrar, Primero en salir).

Cuánto: unidad elemental de tiempo utilizado por el Sistema Operativo para la planificación de procesos. Puede variar para distintos Sistemas Operativos y normalmente consta de una serie de ciclos de reloj.

Dma: acrónimo inglés de 'Direct Memory Access', en castellano, Acceso Directo a Memoria. Consiste en una técnica, implementada bajo un microprocesador especializado, orientada a la transferencia de información desde un dispositivo periférico hasta la memoria principal, sin la utilización para controlarla de la CPU. La CPU solo actúa al principio y fin de dicha transferencia.

Evento o Suceso: un evento o suceso va a consistir en cualquier actuación, ya sea externa o interna que pueda provocar un cambio de estado, a nivel individual en un dispositivo o de forma general en un sistema.

Gestor de Interrupciones: programa encargado de controlar mediante una trampa y redirigir el flujo del programa hacia las rutinas pertinentes que solucionen el problema que provocó la excepción o interrupción.

Lista Enlazada: estructura de datos dinámica residente en memoria principal, la cual puede cambiar de tamaño en tiempo de ejecución. Su acceso es secuencial y está formada por un grupo de elementos en el que cada uno de ellos apunta al próximo.

Pcb: acrónimo inglés de 'Process Central Block', en castellano, Bloque Central de Proceso, el cual almacena información específica sobre un proceso en concreto.

Panificable: todo dispositivo o entidad que es susceptible de ser planificada, en cuanto a tiempo, medios o cualquier otro recurso utilizable por la misma.

Prioridad: orden de importancia relativa de una serie de operaciones planificables. Una operación con una prioridad alta se ejecuta antes que una de baja prioridad, siendo el sentido de mayor a menor prioridad o viceversa.

Procedimiento: rutina o conjunto de instrucciones ejecutable, totalmente independiente, la cual realiza un proceso determinado a partir de una serie de parámetros de entrada, ofreciendo los resultados mediante parámetros de salida.

Signal: operación de continuación sobre un semáforo. Los semáforos serán rutinas, programadas para utilizarlas en programación concurrente.

Tabla de Procesos: estructura de datos que actuando como una tabla de una entrada, almacena los procesos creados en el sistema ya estén listos o bloqueados.

Wait: operación de espera sobre un semáforo. Los semáforos serán rutinas programadas utilizadas en programación concurrente.

Estrategias de Planificación: cada una de las distintas determinaciones que se pueden tomar para realizar un tipo u otro de planificación, basándose en una serie de criterios.

Gasto Extra: gasto de tiempo o cualquier recurso que se deriva de mantener mayor dedicación a tareas de planificación y gestión que a la propia dedicación de tiempo de CPU a los procesos de usuario.

Monopolización de La CPU: un determinado tipo de procesos adquieren los derechos de tiempo de ejecución de forma exclusiva. El resto de procesos no pueden acceder a este recurso imprescindible para avanzar en la ejecución.

Reloj de Interrupciones: reloj que genera interrupciones para marcar los instantes en los que se interrumpe el proceso que se está ejecutando en la CPU, para dar paso a otro elegido entre los procesos de la cola de procesos listos, mediante un algoritmo de planificación.

Tamaño de Cuanto: tamaño en unidades de tiempo, del orden de microsegundos, que dura la unidad mínima de ejecución, el cuánto. De este parámetro depende en gran parte la eficiencia del planificador asignando la CPU a procesos.

Tiempo Real: un sistema de este tipo es aquel que necesita de tiempos de respuesta muy cortos, incluso del orden de microsegundos, en el caso de procesos críticos.

Tiempo de Respuesta: tiempo que se tarda desde que un proceso está listo para ejecutarse hasta que el recurso de la CPU es adquirido por el mismo. De forma general: tiempo total que se tarda en atender la CPU un proceso interactivo ante una petición de servicio.

Acumulador: registro interno de la CPU que recoge los resultados intermedios de las operaciones que se realizan en la ALU.

Asíncrono: término utilizado para especificar la ejecución de distintos procesos de forma independiente unos de los otros respecto al tiempo. En Hardware este concepto es aplicable a dispositivos físicos como la memoria y la CPU, etc.

Cita (Rendezvous): del inglés 'rendezvous', en informática conocido por 'cita'. Tipo de sincronización doble, en el que los dos procesos se sincronizan en un determinado punto de cada uno de los procesos afectados. Uno espera al otro para poder avanzar.

Descriptor de Fichero: número que representa a un fichero lógico con el que se trabaja en lenguaje C. Se obtiene a partir de la asociación que se realiza en una operación de apertura del fichero con su correspondiente nombre físico.

Espera Bloqueada: tipo de 'espera' en primitivas para solucionar el problema de la exclusión mutua y sincronización mediante el mantenimiento de colas de procesos bloqueados hasta que se da una cierta condición y pasen al estado de listos. Se da en mecanismos software como los semáforos y monitores.

Hilo: cada una de las unidades de asignación de un proceso, de esta forma cada proceso (elemento que posee recursos), puede mantener varias unidades de asignación de CPU en una misma ejecución. Este concepto se conoce también como 'thread', hebra o proceso ligero.

Multiprocesador: sistema informático basado en la ejecución de programas mediante la utilización de varios procesadores trabajando de forma simultánea. Pueden disponerse físicamente de varias formas: en serie, paralelo,

Llamadas por Referencia: llamada a un procedimiento de forma que, mediante parámetros el objeto que se envía al mismo es el espacio de direcciones del objeto original, el cual guarda el procedimiento llamador. De esta forma cualquier cambio sobre el mismo será más difícil de implementar debido al cruce de direcciones.

Llamada por Valor: llamada a un procedimiento de forma que, con la ayuda de parámetros el objeto que se envía al mismo es una copia del original que guarda el procedimiento llamador. De esta forma cualquier cambio sobre el mismo afectará solo a la copia.

Protocolos: estándar de comunicación entre distintos dispositivos físicos o procesos mediante el cual se regulan los mecanismos y políticas a seguir para llevar a cabo una transmisión perfecta de datos.

Recurso: objetos o dispositivos que son utilizados por una computadora, para poder realizar todos los trabajos y tareas que se requieren a partir de las peticiones que van realizando los procesos del sistema. Entre otros en una computadora tenemos como recursos los siguientes: memoria principal, dispositivos de memoria secundaria, y cualquier dispositivo periférico direccionable. Además estos pueden ser o no compartibles.

Red de Máquinas Heterogéneas: conjunto de ordenadores de distinta naturaleza o composición interna debido a sus distintas arquitecturas.

Semáforo: Un semáforo es un mecanismo software mediante el cual consigo la sincronización entre procesos concurrentes. Su implementación estará basada en dos elementos: un entero y un puntero asociado a una cola (que puede ser nulo). Está controlado mediante una serie de señales conocidas como WAIT y SIGNAL.

Tubo o Tubería: mecanismo software para solucionar problemas de comunicación y sincronización entre distintos procesos, mediante la implementación de un canal software de comunicación asíncrono.

Lista Circular: estructura de datos dinámica residente en memoria principal, derivada de las listas, la cual puede cambiar de tamaño en tiempo de ejecución. Su acceso es secuencial y tiene como característica que el último elemento de la misma mantiene un enlace al primero, uniendo a ambos.

Tiempo de Búsqueda: tiempo que transcurre desde que se recibe la orden de transferencia hasta que la cabeza lectora se posiciona sobre el disco donde buscamos los datos.

Tiempo de Latencia: tiempo que transcurre desde que se da la orden de posicionamiento sobre la cabeza lectora al cabezal que soporta el disco, hasta que está colocado en su lugar correcto donde están los datos se quiere transmitir.

Tiempo de Transferencia: tiempo que se tarda en transmitir una serie de datos desde el disco a memoria principal, desde que los datos son encontrados por la cabeza lectora hasta que finaliza la transmisión.

Trampa de Fallo de Página: interrupción provocada por un acceso a una página en memoria principal, la cual no estaba en la misma debido a que una administración virtual de la memoria provocó su traslado a disco, mediante un intercambio de páginas.

Trampa: método consistente en atrapar u o capturar una interrupción mediante la comprobación de una condición en particular en un programa en ejecución, para procurar después la ejecución de la rutina correspondiente que resuelva dicha interrupción.

Referentes

DICCIONARIO DE COMPUTACIÓN⁴¹

ALAN FREEDMAN

MC-GRAW HILL (5ª Edición)

<http://www.di.ujen.es/~lina/TemasSO/glosario/GLOSARIO.htm>

5.4. Bibliografía

- ◆ Tanenbaum S. Andrew Sistemas operativos modernos, PRENTICE-HALL Segunda Edición Español
- ◆ Stallings William Sistemas operativos, PRENTICE-HALL Tercera Edición 2001
- ◆ García Carretero Sistemas operativos una visión aplicada, Primera Edición 2001
- ◆ Silberschaz Sistemas Operativos, PRENTICE HALL Sexta Edición 2002
- ◆ Flynn Mchoes Sistemas Operativos.
- ◆ Medellín. Matemáticas y sistemas discretos. Recuperado el 25 de marzo de 2011, de <http://docencia.udea.edu.co/SistemasDiscretos/contenido/principal.html>
- ◆ Forero, A. (2004). Lógica proposicional. Recuperado el 25 de marzo de 2011, de <http://elcentro.uniandes.edu.co/cr/mate/estructural/libro/estructural/node3.html>
- ◆ <http://exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos/SO0.htm>
- ◆ http://es.wikibooks.org/wiki/Sistemas_operativos/Clasificaci%C3%B3n
- ◆ <http://es.kioskea.net/contents/systemes/sysintro.php3>
- ◆ [http://es.wikipedia.org/wiki/Proceso_\(inform%C3%A1tica\)](http://es.wikipedia.org/wiki/Proceso_(inform%C3%A1tica))
- ◆ <http://exa.unne.edu.ar/depar/areas/informatica/SistemasOperativos/MonogSO/INTSI02.htm#raul>
- ◆ <http://sopa.dis.ulpgc.es/so/teoria/pdf/>
- ◆ <http://www.augcyl.org/?q=glol-intro-sistemas-distribuidos>
- ◆ <http://laurel.datsi.fi.upm.es/docencia/asignaturas/sod>
- ◆ Andrew S. Tanenbaum: "Sistemas Operativos Distribuidos", Prentice Hall (1996).
- ◆ George Coulouris, Jean Dollimore, Tim Kindberg: "Distributed Systems, Concepts and Design", Addison-Wesley (1994).