Manejo de memoria en Sistemas Operativos

Miguel Angel Astor Romero

Versión 1 - 28 de septiembre de 2016

Introducción

En la actualidad toda computadora de propósito general funciona según la arquitectura de Von Neumann, centrada en el concepto de programa almacenado. Según esta arquitectura, una computadora se compone de tres elementos fundamentales [1]. El primero de estos es el CPU (*Central Processing Unit* - Unidad Central de Procesos) encargado de ejecutar instrucciones provistas por el usuario en un programa. El segundo elemento es la memoria, la cual es una estructura lineal de bits (dígitos binarios, la unidad mínima de información procesable) donde se almacenan tanto los programas como los datos sobre los que estos operan. El último elemento que compone a la computadora son los dispositivos de entrada/salida que permiten al usuario el introducir datos y programas a la misma y a su vez el recibir información generada por la computadora. Esta arquitectura puede verse en la Figura 1.

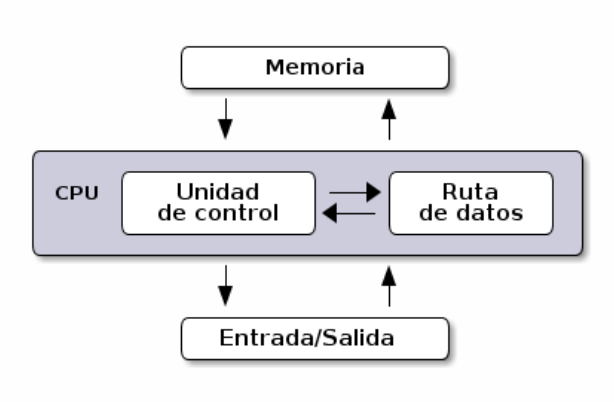

Figura 1: . Figura recuperada de [2].

La memoria se presenta entonces como un componente fundamental el cual debe ser gestionado correcta y eficientemente para garantizar el correcto funcionamiento de la computadora. Originalmente, en la época de los *mainframes* y los sistemas de procesamiento por lotes , esta tarea era responsabilidad del programador de aplicaciones [3]. Sin embargo, con el avance de la tecnología y la introducción de conceptos como la multiprogramación y el

tiempo compartido (donde una computadora permite la ejecución de múltiples programas provistos por uno o más usuarios, simulando la ejecución concurrente de los mismos), se hizo necesario por razones de seguridad y comodidad para los usuarios y programadores de aplicaciones el delegar la tarea de manejo y gestión de la memoria al sistema operativo [3].

Posteriormente con el avance de los sistemas operativos distribuidos y la introducción del esquema de diseño de imagen de sistema único, se hizo igualmente necesario el desarrollo de mecanismos que permitieran a estos sistemas el gestionar la memoria de múltiples computadoras conectadas a la red de forma que, a ojos de los programadores y usuarios, estas parecieran proveer un espacio de memoria global central, haciendo transparente el uso del mismo [4].

El resto de este documento se estructura de la siguiente manera. En la Sección 1 se detallan los distintos esquemas de manejo de memoria en sistemas operativos centralizados de propósito general, haciendo particular énfasis en los mecanismos y algoritmos utilizados para el manejo de memoria caché. La Sección 2 presenta las distintas técnicas de gestión de memoria principal utilizadas en los sistemas operativos distribuidos. Finalmente se presentan las conclusiones del documento.

1. Memoria en sistemas operativos centralizados

La memoria de las computadoras se organiza en una jerarquía la cual ubica distintos tipos de memoria en niveles según su costo, velocidad de acceso y cercanía al CPU [3]. Esta jerarquía puede observarse en la Figura 2. En esta jerarquía, los registros de CPU son pequeñas memorias sumamente rápidas las cuales se encuentran físicamente en el chip del CPU y se utilizan para guardar los datos y las instrucciones específicas sobre las que está operando el procesador en un momento dado. Luego está la memoria principal comúnmente conocida como memoria RAM (*Random Access Memory* - Memoria de Acceso Aleatorio) la cual se utiliza para almacenar datos y programas.

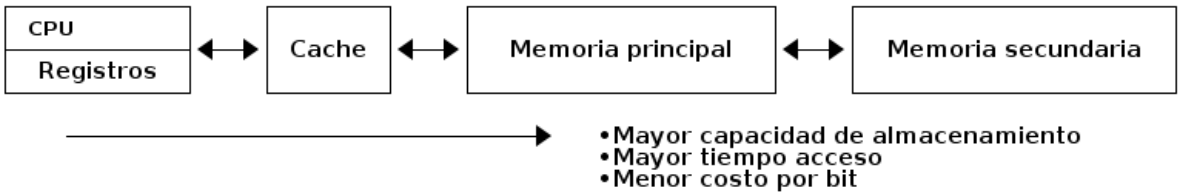

Figura 2: Jerarquía de la memoria. Figura recuperada de [5].

Entre el CPU y la memoria principal pueden existir uno o más niveles de memorias caché que sirven para almacenar conjuntos de datos mayores que los que pueden caber en los registros pero que pueden ser utilizados por el procesador próximamente [3]. Las memorias caché mejoran el rendimiento del sistema al reducir la cantidad de transferencias entre la

memoria principal y el procesador [5]. Finalmente está la memoria secundaria la cual se utiliza para almacenar datos y programas de forma persistente puesto que las demás memorias de la jerarquía son volátiles, es decir, pierden su contenido al ser apagadas [3].

El sistema operativo es el encargado de gestionar el uso de las memorias en todos los niveles de esta jerarquía, acción que realiza con un componente conocido como el *memory manager* (manejador de memoria) [6]. El *memory manager* puede seguir múltiples estrategias para gestionar la memoria las cuales son detalladas a continuación en esta Sección.

1.1. Memoria simple

El mecanismo más simple de manejo de memoria es sencillamente no manejar la memoria en absoluto [6]. De esta forma se presenta a los programas de usuario con la totalidad de la memoria física de la computadora la cual pueden compartir con el sistema operativo. En este esquema cada acceso a la memoria se basa en referencias o direcciones de memoria absolutas ubicadas entre 0 y la memoria máxima disponible.

En este esquema cada acceso a memoria es absoluto y como tal no provee ninguna garantía de protección sobre la memoria, siendo posible entonces que un programa modifique memoria utilizada por otro programa o incluso por el sistema operativo mismo [3]. Este sistema era utilizado en las primeras computadoras personales las cuales utilizaban sistemas operativos incapaces de ejecutar múltiples programas simultáneamente. Actualmente este mecanismo se utiliza en los sistemas embebidos aunque está completamente desfasado en los sistemas de propósito general [3].

Para resolver estos problemas se desarrolló el concepto de espacios de memoria según el cual cada proceso cree que tiene disponibilidad y control absoluto de una cantidad de memoria contigua de cierto tamaño que asume es la memoria física [3]. Para que los procesos puedan funcionar se hace necesario entonces el poder relacionar este espacio de memoria ficticio con la memoria real de la computadora. Esto se logra asociando a cada proceso dos direcciones de memoria conocidas como el registro base y el registro límite respectivamente [7]. De esta manera, cada vez que un proceso quiera realizar un acceso a memoria generará una dirección lógica o virtual ubicada entre 0 y el máximo del espacio de direcciones que se le ha asignado. Esta dirección lógica es utilizada por un componente del CPU conocido como la MMU (*Memory Management Unit* - Unidad de Manejo de Memoria) encargada de traducir cada dirección lógica por su correspondiente dirección real o física, sumando el valor del registro base a las direcciones generadas por el proceso [7]. Una extensión a este mecanismo es permitir que los espacios de memoria aumenten o disminuyan de tamaño de forma dinámica siempre que haya memoria física disponible [3], acción que realiza el sistema operativo.

El uso de espacios de memoria permite adicionalmente al sistema operativo el dar un nivel de protección a los procesos evitando que un proceso pueda modificar el espacio de memoria de otro. Esto se logra verificando que cada dirección generada por un proceso se ubique entre las direcciones especificadas por el registro base y el registro límite. La MMU es la encargada de realizar esta verificación [7]. Igualmente, los espacios de memoria proveen la posibilidad de mover un proceso de una región de memoria a otra mediante un procedimiento conocido

como relocalización [7], e incluso permiten el poder suspender un proceso, desalojarlo de la memoria y almacenarlo en la memoria secundaria para reanudarlo posteriormente. Este último procedimiento se conoce como intercambio [7].

1.2. Segmentación

La segmentación es una extensión al esquema de los espacios de memoria en el cual a cada proceso se le pueden asignar múltiples espacios de direcciones independientes llamados segmentos [3]. Cada segmento es entonces un espacio de direcciones lineal cuyas direcciones comienzan en 0 hasta un máximo. El tamaño de un segmento ubicado en memoria física puede variar de forma dinámica e independientemente de los demás segmentos utilizados por el proceso al que pertenece [3]. Para poder utilizar segmentos un proceso debe generar direcciones lógicas divididas en dos partes, la primera siendo un indicador del segmento a utilizar y la segunda un desplazamiento dentro del segmento. Por ejemplo, en un sistema que utilice direcciones de 32 bits se pueden destinar los primeros 7 bits para indicar el número de segmento y los restantes 25 bits como desplazamiento. De esta forma el sistema mencionado puede soportar hasta $2^7 = 128$ segmentos por proceso, los cuales pueden ser de a lo sumo $2^{25} = 33554432$ bytes cada uno.

En este esquema cada segmento es una entidad lógica que los programadores pueden manipular directamente con conocimiento de la existencia de las mismas. Esto permite la posibilidad de utilizar segmentos para almacenar código o estructuras de datos con gran flexibilidad. Por lo general cada segmento se destina únicamente a almacenar datos o código de forma exclusiva. Gracias a esto es posible entonces el asignar distintos niveles de protección a los diferentes segmentos, permitiendo que por ejemplo los segmentos con estructuras de datos puedan ser leídos o escritos pero no ejecutados, mientras que los segmentos con código solo pueden ser leídos y ejecutados pero no modificados [3].

Para hacer uso de la segmentación, el sistema operativo debe mantener una estructura por cada proceso conocida como la tabla de segmentos [7]. Esta tabla contiene por cada segmento que puede tener un proceso los siguientes datos: un bit para indicar si el segmento es válido, datos sobre los permisos de uso del segmento (lectura, escritura y ejecución), la dirección inicial del segmento en memoria principal (registro base) y la dirección final del segmento (registro límite). El número de segmento de las direcciones lógicas se utiliza para indexar esta tabla.

Como los segmentos son de tamaño variable es posible acomodar estos a los datos que contengan de forma que no exista espacio desperdiciado dentro de un segmento, espacio conocido como fragmentación interna [3]. Sin embargo, dado que los segmentos pueden ser creados y destruidos de forma dinámica, es posible que se formen espacios no utilizados entre los segmentos los cuales pueden llegar a ser demasiado pequeños como para poder almacenar un segmento. Este problema se conoce como fragmentación externa [3]. La Figura 3 muestra un ejemplo de la formación de fragmentación externa en un sistema con segmentación. La fragmentación externa puede combatirse con una estrategia llamada compactación de la memoria con la cual los segmentos son desplazados hacia un extremo de la memoria física de forma que el espacio libre se ubique completamente en el extremo opuesto de la misma

[3]. La Figura 3 ilustra el resultado de un proceso de compactación de memoria.

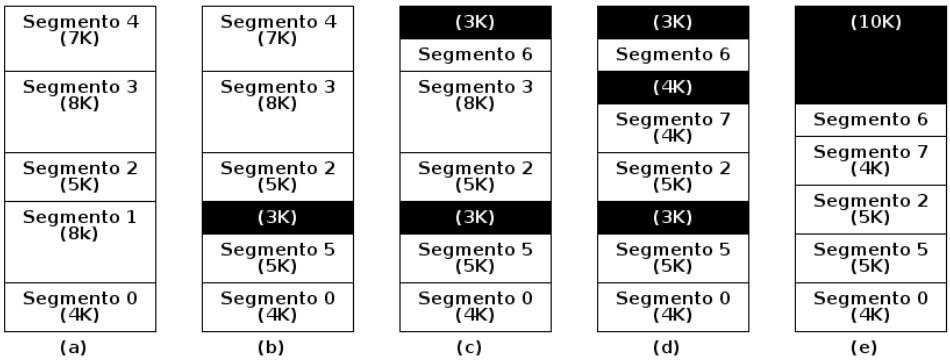

Figura 3: (a)-(d) Formación de la fragmentación externa. (e) Compactación de la memoria. Figura recuperada de [3].

Para poder ubicar segmentos en la memoria principal se puede utilizar cualquiera de los siguientes algoritmos, conocidos en conjunto como algoritmos de ajuste [7].

Primer Ajuste El segmento se ubica en el primer espacio libre contiguo de tamaño suficiente para almacenarlo. Cada ejecución del algoritmo revisa los espacios libres desde el primero hasta que encuentre uno utilizable.

Mejor Ajuste El segmento se coloca en el espacio libre contiguo de menor tamaño capaz de almacenar el segmento. Implicar revisar todos los espacios libres disponibles.

Peor Ajuste Equivalente al algoritmo de mejor ajuste, pero utiliza el espacio libre más grande en lugar del de menor tamaño.

Siguiente Ajuste Equivalente a primer ajuste pero almacena la ubicación del siguiente espacio libre disponible al escogido por la invocación anterior del algoritmo para poder comenzar la siguiente búsqueda desde esa posición.

Knuth demostró en [8] que de estos algoritmos Primer Ajuste es el que produce menor cantidad de fragmentación externa a largo plazo mientras que Mejor Ajuste es el que produce la mayor cantidad de fragmentación.

1.3. Memoria virtual y paginación

El mecanismo de memoria virtual fue inventado en 1961 como una forma de permitir que un proceso fuera capaz de utilizar un espacio de memoria que potencialmente puede ser mayor que la totalidad de la memoria física disponible [6]. Esto se logra dividiendo el espacio de memoria del proceso en varias partes que pueden ser cargadas en memoria física

o intercambiadas a la memoria secundaria cuando no son necesarias. De esta forma, si un proceso necesita utilizar una parte de si mismo que no se encuentra en memoria física deberá bloquearse por entrada/salida mientras esta es cargada a la memoria, permitiendo que otro proceso pueda ejecutarse mientras tanto [6].

La técnica de memoria virtual más común se conoce como paginación [6]. Mediante este esquema, el espacio de memoria de un proceso se divide en bloques de tamaño fijo llamados páginas las cuales pueden corresponderse con bloques del mismo tamaño en memoria principal conocidos como marcos de página. Idealmente el espacio de memoria virtual de un proceso será del tamaño de la memoria máxima direccionable, independientemente de si existe suficiente memoria física para almacenarlo completamente o no [6]. Cuando un proceso necesita descargar parte de su memoria al almacenamiento secundario, o viceversa lo hará siempre en grupos de páginas.

Para hacer la correspondencia entre una página y un marco de página el proceso genera direcciones virtuales divididas en dos partes, la primera correspondiente al número de página y la segunda a un desplazamiento dentro de la misma. El número de página en una dirección virtual se utiliza como índice de una estructura de datos del sistema operativo llamada tabla de páginas de la cual existe una por cada proceso en el sistema [6]. Esta tabla contiene una serie de entradas cada una de las cuales contiene por lo menos un bit indicando si la página en cuestión actualmente tiene correspondencia con un marco de página, un bit para indicar si la página ha sido referenciada por el proceso, un bit para indicar que la página fue modificada desde que fue cargada en memoria, además de un campo donde se guarda la dirección inicial del marco de página en la cual se encuentra cargada la página si aplica.

En caso de que el número de páginas disponibles a los procesos sea muy grande entonces el tamaño de la tabla de páginas será considerablemente grande también. Para evitar desperdiciar memoria principal en almacenar muchas tablas de páginas de gran tamaño se utiliza un mecanismo conocido como tablas de páginas multinivel con el cual una tabla de páginas de primer nivel tiene entradas que referencias tablas de páginas de segundo nivel las cuales pueden llevar a entradas de página o incluso a tablas de tercer nivel y así sucesivamente [6]. De esta forma es posible cargar solamente las tablas de segundo nivel necesarias, intercambiando las demás a la memoria secundaria. Sin embargo, para soportar este mecanismo se deben dividir las direcciones virtuales en tres partes (índice de primer nivel, índice de página, desplazamiento) para tablas de dos niveles, o cuatro partes para tablas de tres niveles y así sucesivamente [6].

La búsqueda de la entrada en la tabla de páginas y la traducción de direcciones se realiza en la MMU la cual posee un dispositivo especializado llamado TLB (*Translation Lookaside Buffer* - Buffer de Búsqueda de Traducciones) el cual permite realizar la búsqueda en una tabla de páginas de forma rápida sin tener que generar accesos adicionales a la memoria para realizarla [6]. Si la MMU no puede encontrar una correspondencia entre una página y un marco de página entonces no será capaz de realizar la traducción de direcciones. Ante esta situación la MMU genera una trampa conocida como fallo de página, la cual le indica al sistema operativo que es necesario cargar la página solicitada a la memoria física para crear la asociación correspondiente con un marco de página.

Si existen marcos libres que no posean asociación con ninguna página entonces una página que deba cargarse en memoria principal puede colocarse sin problemas en cualquiera de estos marcos. Sin embargo, si todos los marcos de página están ocupados será necesario entonces el intercambiar uno de estos a almacenamiento secundario para poder colocar en él la pagina en cuestión. Para realizar esto se utiliza uno de los siguientes algoritmos de sustitución de páginas [6].

Algoritmo óptimo De entre todas las páginas cargadas en memoria se descarga la que pasará más tiempo sin ser referenciada. Este algoritmo es imposible de implementar pero provee una cota superior con la cual puede compararse el desempeño de los demás algoritmos, siendo el mejor el que más se aproxime al rendimiento del algoritmo óptimo.

NRU (*Not Recently Used* - No Recientemente Utilizada) Se agrupan las páginas en las siguientes cuatro categorías ordenadas, utilizando sus bits de referencia y modificación. Se descarga una página aleatoria de la categoría de menor número no vacía.

1. Sin referenciar y sin modificar.
2. Sin referenciar, modificada.
3. Referenciada, sin modificar.
4. Referenciada y modificada.

FIFO (*First-In First-Out* - Primero en Entrar Primero en Salir) Se descarga la página con más tiempo en la memoria principal. Para implementar este algoritmo el sistema operativo mantiene una cola en la cual las páginas nuevas se insertan por la parte de atrás mientras que siempre se reemplaza la página al comienzo de la misma.

Segunda Oportunidad Equivalente a FIFO, pero si la página a reemplazar ha sido referenciada por el proceso se le permite permanecer en memoria, reemplazando en su lugar a la página más antigua que no haya sido referenciada. Una página que se permite permanecer en memoria es marcada como no referenciada y colocada al final de la cola de páginas del sistema.

Reloj Equivalente a Segunda Oportunidad pero utiliza una cola circular en lugar de una cola enlazada para evitar tener que reinsertar las páginas que se ha decidido mantener en memoria.

LRU (*Least Recently Used* - Menos Recientemente Utilizada) Se descarga la página que haya pasado la mayor cantidad de tiempo sin haber sido referenciada. Requiere de hardware especializado para ser implementado eficientemente.

NFU (*Not Frequently Used* - No Utilizada Frecuentemente) Aproximación a LRU en software. Cada marco de página tiene un contador asociado que inicialmente vale 0. En cada interrupción de reloj se suma el valor del bit de referencia de la página asociada a cada marco al contador. Se reemplaza la página que tenga el contador más bajo.

Envejecimiento Mejora de NFU. En cada ciclo de reloj se aplica un desplazamiento lógico a la derecha al contador de cada marco, luego se coloca el valor del bit de referencia en el bit más a la izquierda del contador.

1.4. Memoria caché

Como se mencionó al inicio de la Sección 1, entre la memoria principal y el procesador pueden existir uno mas niveles de memoria caché que se utiliza para evitar que el CPU tenga que solicitar datos directamente a la memoria principal, que es considerablemente más lenta que el procesador [9].

Las memorias caché se basan en un principio conocido como localidad de referencia, el cual establece que es más probable que en un momento dado un programa haga referencias cercanas a un conjuntos de datos reducido de la memoria más que a la totalidad de la misma o incluso que referencias aleatorias [9]. De esta manera es posible mantener estos conjuntos de datos más cerca del CPU, de forma que este pueda obtener acceso rápido a los mismos sin tener que hacer solicitudes directamente a la memoria principal.

El principio de localidad se presenta en dos variantes [9]. La primera de estas se llama localidad espacial y establece que es más probable el acceso a memoria ubicada espacialmente cerca. Por ejemplo, si un proceso ha solicitado leer las direcciones de memoria 100, 101 y 102 en sus últimas referencias a memoria, entonces es más probable que solicite la dirección 103 a continuación que por ejemplo solicite la dirección 1000. La segunda variante se conoce como localidad temporal e indica que es más probable que un proceso solicite nuevamente datos recientemente utilizados que datos que no han sido utilizados en mucho tiempo. Cabe resaltar que ambas formas del principio de localidad consisten en observaciones experimentales, no son leyes generales que se cumplen en todo momento [9].

1.4.1. Caché del procesador

Siguiendo el principio de localidad de referencia las memorias caché del procesador se organizan en bloques de tamaño fijo conocidos como lineas de caché. Cuando el procesador genera una referencia a memoria física, se busca esta dentro de la memoria caché. Si existe se toma de allí, pero en caso de que no exista entonces se genera una trampa llamada fallo de caché que le indica al sistema operativo que debe cargar una linea de caché con la referencia solicitada de la memoria principal [9]. Por ejemplo, si una linea de caché es de 64 bytes y el procesador solicita la dirección de memoria física 260, entonces el sistema operativo deberá cargar los datos de las direcciones 256 a la 319 a la caché los cuales constituyen una linea de caché.

1.4.2. Caché de páginas

Se conoce por caché de páginas a una estrategia que se utiliza para acelerar el acceso a archivos y otros datos del almacenamiento secundario haciendo uso del espacio libre de la memoria principal [7]. Con esta estrategia, los marcos de página se pueden utilizar tanto

para almacenar páginas de procesos en ejecución como bloques leídos del almacenamiento secundario los cuales se colocan en páginas asociadas a los procesos pero que no forman parte de su espacio de direcciones virtual. Este esquema es también conocido como memoria virtual unificada [7].

1.5. Algoritmos de sustitución de caches

Para realizar la sustitución de lineas en una memoria caché se pueden utilizar cualquiera de los algoritmos de sustitución de páginas mencionados en la Sección 1.3. Sin embargo, el más utilizado es el algoritmo LRU descrito en la misma Sección, el cual es implementado en hardware por la MMU [9].

2. Memoria en sistemas operativos distribuidos

Se entiende por sistema distribuido a "una colección de computadoras independientes que parecen ser una única computadora a los usuarios del sistema" [10]. Implícito en esta definición se encuentra el hecho de que estas computadoras independientes están conectadas de alguna manera mediante una red. Un sistema operativo distribuido es entonces el sistema operativo encargado de gestionar estas computadoras para lograr el objetivo de ocultar la naturaleza distribuida del sistema a los usuarios.

La idea de que un sistema distribuido debe aparecer ante los usuarios como un sistema centralizado se conoce como imagen de sistema único [4] y se intenta alcanzar incorporando distintos principios de transparencia en el diseño de los sistemas operativos distribuidos [10], principios que son utilizados por los desarrolladores de aplicaciones. Los principios de transparencia de los sistemas distribuidos son los siguientes [10]:

Transparencia de ubicación Los usuarios no pueden distinguir donde se encuentran los recursos.

Transparencia de migración Los recursos pueden moverse por el sistema sin cambiar de nombre.

Transparencia de replicación Los usuarios no pueden saber cuantas copias de un recurso existen.

Transparencia de concurrencia Múltiples usuarios pueden compartir recursos automáticamente.

Transparencia de paralelismo Se pueden realizar actividades en paralelo sin que el usuario se entere.

La gestión de la memoria en los sistemas operativos distribuidos tiene como objetivo el apoyar los principios de transparencia de ubicación, migración y replicación [10].

2.1. Memoria compartida

Dado que los sistemas operativos distribuidos tienen como objetivo el presentar un conjunto de computadoras débilmente acopladas como si fueran un sistema centralizado, conviene entonces el estudiar el funcionamiento de la memoria en estos sistemas. En particular, cuando un sistema centralizado se compone de múltiples procesadores presenta múltiples complicaciones de diseño e implementación compartidas por los sistemas distribuidos [3].

Los sistemas centralizados con dos o más procesadores se conocen como sistemas multiprocesador de memoria compartida [3]. En estos sistemas existe una única memoria principal global la cual puede ser utilizada por todos los procesadores del sistema, cada uno de los cuales posee su propia memoria caché (de uno o más niveles) y sus respectivos conjunto de registros [3] siguiendo la misma jerarquía de memoria establecida en la Figura 2.

Los sistemas multiprocesador de memoria compartida se pueden organizar según distintas arquitecturas [3]. Una arquitectura común en las computadoras personales es la arquitectura de bus, en la cual existe un canal de comunicación compartido entre los procesadores y la memoria principal llamado bus del sistema. Esta arquitectura puede observarse en la Figura 4. Con un bus de sistema es necesario utilizar un mecanismo de sincronización entre los distintos procesadores cuando estos quieran utilizar la memoria principal para evitar problemas causados por la colisión de datos en el bus [3]. Otras arquitecturas utilizadas en estos sistemas utilizan múltiples lineas de comunicación entre los procesadores y la memoria mediante circuitos de conmutación de datos [3].

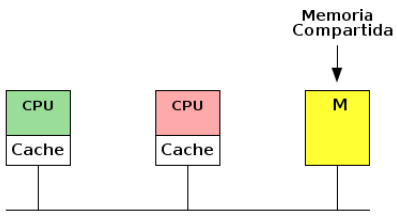

Figura 4: Arquitectura de bus de sistema. Figura recuperada de [3].

En un sistema de memoria compartida cualquier procesador puede leer o modificar el contenido de cualquier ubicación de la memoria en cualquier momento, restringidos por los mecanismos de protección de acceso implementados por el sistema operativo [3]. Para evitar los retrasos introducidos por el mecanismo de sincronización del bus o el circuito de conmutación son de vital importancia las memorias caché de los procesadores, las cuales funcionan como se describió en la Sección 1.4.

La existencia de múltiples memorias caché en los distintos procesadores introduce un problema adicional que se presenta cuando varios procesadores contienen copias de un mismo dato de la memoria principal en sus respectivas memorias caché. Si en un momento dado un procesador modifica dicho dato y luego otro procesador desea leer el mismo dato, este debería utilizar el valor modificado que el procesador anterior mantiene en su caché, en lugar de su copia des-actualizada. De la misma forma, si un procesador que no posee dicho dato en su

memoria caché necesita leer o escribir ese dato, deberá hacerlo sobre la copia modificada en lugar de sobre la memoria principal [3]. Este problema se resuelve introduciendo protocolos de coherencia de caches, los cuales se describen en la Sección 2.5. Este problema también se presenta en los sistemas de archivos distribuidos [11].

2.2. Memoria distribuida

En contraposición con los sistemas multiprocesador de memoria compartida, un sistema multicomputador de memoria distribuida es un sistema distribuido en el cual cada computadora posee su propia memoria privada la cual es inalcanzable a las demás computadoras del sistema [10]. Un ejemplo de este tipo de sistemas puede verse en la Figura 5. Los sistemas de memoria distribuida se caracterizan por no proveer los principios de transparencia mencionados anteriormente [10], razón por la cual se desarrollaron los sistemas de memoria compartida distribuida descritos en la siguiente Sección.

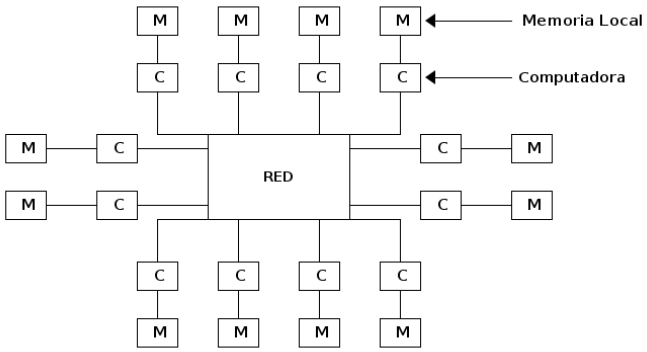

Figura 5: Sistema de memoria distribuida. Figura recuperada de [3].

2.3. Memoria compartida distribuida

La memoria compartida distribuida o DSM (*Distributed Shared Memory*) es un mecanismo de abstracción que simula la existencia de una memoria compartida sobre un sistema de memoria distribuida [12]. De esta forma, un conjunto de procesos que ante la ausencia de DSM necesitarían compartir datos mediante mecanismos de comunicación explícitos como el paso de mensajes pueden realizar esta acción de forma transparente [12]. La Figura 6 ilustra el diseño general de un sistema con DSM.

En un sistema con DSM, esta se presenta como un ente lógico que representa una memoria global compartida cuyo tamaño es igual a la suma de las memorias locales de los nodos que la componen. Los procesos utilizan la DSM como parte de su espacio de memoria, pudiendo utilizar datos ubicados físicamente en computadoras distintas de forma transparente.

Para implementar un sistema de memoria compartida distribuida se utiliza uno de los siguientes tres enfoques [12]:

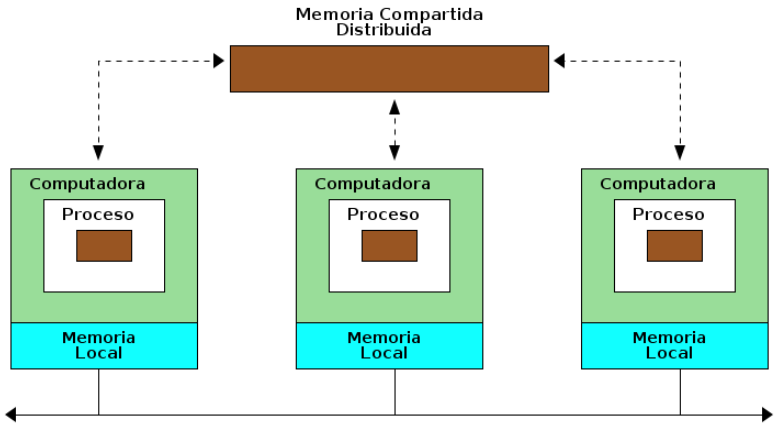

Figura 6: Memoria compartida distribuida. Figura recuperada de [12].

Basado en hardware Se utiliza hardware especializado que extiende la MMU de los procesadores del sistema con mecanismos que permiten resolver direcciones de DSM y realizar automáticamente la comunicación necesaria para el intercambio de datos.

Basado en paginación El nucleo del sistema operativo distribuido presenta la DSM como un espacio de memoria virtual que lógicamente funciona como se describió en la Sección 1.3, utilizando intercambio de páginas con paso de mensajes entre las distintas computadoras del sistema en lugar de intercambiar las páginas a almacenamiento secundario.

Basado en bibliotecas La DSM se implementa como un mecanismo a nivel del entorno de tiempo de ejecución del lenguaje de programación, siendo el compilador o el intérprete del lenguaje los encargados de proveer la transparencia en los accesos a memoria.

La DSM basada en hardware solo se ha utilizado en sistemas multiprocesador experimentales con soporte de red [12]. Los otros dos enfoques son los más estudiados debido a que estos no imponen restricciones particulares sobre el mecanismo de DSM y no poseen los costos adicionales implicados por una solución con hardware de propósito específico [12].

Con respecto al contenido de la DSM, esta se puede organizar de tres formas distintas [12]:

Orientada a bytes La DSM se utiliza como un arreglo lineal de bytes.

Objetos compartidos La DSM provee un espacio para almacenar objetos entendidos como instancias de clases en programación orientada a objetos.

Datos inmutables La DSM se utiliza para almacenar datos que una vez han sido escritos no pueden ser modificados. En lugar de esto, los datos deben ser reemplazados si se desea almacenar alguna modificación aplicada a los mismos.

Visualizar la DSM como un arreglo de bytes provee absoluta flexibilidad con respecto a su uso [12] y se utiliza como una memoria virtual normal. En cambio, los mecanismos de objetos compartidos permite al sistema operativo distribuido o al lenguaje de programación el incorporar mecanismos de serialización de datos y sincronización directamente a la DSM, implementados como operaciones sobre objetos [12]. Adicionalmente, el mecanismo de datos inmutables evita completamente los problemas de consistencia descritos en la siguiente Sección a cambio de pérdida de transparencia [12].

2.4. Modelos de consistencia

Para disminuir la cantidad de comunicaciones utilizadas para transferir datos entre computadoras, los mecanismos de DSM utilizan técnicas de memoria caché en la memoria principal de las computadoras que componen el sistema [12]. Esto introduce el problema mencionado en la Sección 2.1 con respecto a la replicación de los datos el cual se conoce como problema de consistencia de datos.

Para resolver este problema se introduce el concepto de modelo de consistencia el cual se define como un contrato entre los procesos y el mecanismo de almacenamiento de datos [4]. Este contrato establece que pueden esperar los procesos con respecto a los datos si se siguen ciertas reglas al momento de leer o modificar los mismos. En otras palabras, los modelos de consistencia restringen los valores que se pueden esperar de un conjunto de datos dependiendo del orden en que se realizen las operaciones de lectura y escritura [4].

Los modelos de consistencia pueden definirse desde dos puntos de vista, uno centrado en los datos y otro centrado en el cliente (entendido como un proceso que necesita leer o escribir un dato). Los modelos de consistencia centrados en los datos son los listados a continuación [4, 10]. Es posible definir modelos de consistencia adicionales si se permite el uso explícito de mecanismos de sincronización [4].

Consistencia estricta La lectura de un dato específico siempre retornará el valor más reciente escrito a ese dato.

Consistencia secuencial Establece que si un conjunto de procesos generan una serie de operaciones de lectura o escritura sobre datos, entonces cualquier intercalamiento de estas operaciones será una secuencia válida pero las operaciones generadas por un proceso particular serán vistas por los demás procesos en el orden exacto en que fueron generadas.

Consistencia causal Se dice que dos operaciones sobre un dato tienen una relación causal si el valor escrito por una operación depende del valor escrito por la otra de alguna manera. En cambio, dos operaciones que no poseen una relación de causalidad se dice que son concurrentes. Este modelo de consistencia establece entonces que todas las operaciones con relación de causalidad serán vistas en el orden necesario para mantener la causalidad, pero las operaciones concurrentes pueden verse en cualquier orden.

Consistencia PRAM Las escrituras emitidas por un proceso serán vistas en el orden en que son emitidas por ese proceso pero las operaciones de todos los procesos pueden verse en cualquier orden.

Centrados en el cliente se pueden definir los siguientes modelos de consistencia [4].

Consistencia momentánea Las actualizaciones a los datos se propagan lentamente, pudiendo un cliente obtener dos valores diferentes en lecturas sucesivas realizadas despues de que otro proceso realizó una escritura. Asume que las modificaciones a los datos son mucho menos frequentes que las lecturas.

Lectura monotónica Si un proceso lee un valor espécifico de un dato, cualquier lectura sucesiva retornará ese mismo valor o un valor más reciente.

Escritura monotónica Cuando un proceso realiza una operación de escritura sobre un dato, se garantiza que cualquier escritura anterior realizada por el mismo proceso a otra copia del dato en una ubicación debe haber completado, es decir, ya se ha propagado a la copia que se está modificando actualmente.

Lea sus escrituras Cuando un proceso realizar una operación de escritura sobre un dato, cualquier lectura postrerior generada por el mismo proceso verá dicha escritura independientemente de sobre que copia del dato se realize la operación de lectura.

Las escrituras siguen a las lecturas Cualquier escritura a un dato que siga a una lectura previa originada por el mismo proceso será aplicada únicamente sobre el valor leido o sobre otro más reciente.

Los modelos de consistencia son implementados a nivel del sistema operativo distribuido o del lenguaje de programación, dependiendo de cual de estos implemente el mecanismo de DSM, mediante un protocolo de consistencia [4].

2.5. Caches distribuidas

Cuando se poseen múltiples procesadores o computadoras cada una con su propia memoria caché y se permite el acceso a datos compartidos, se presentan entonces problemas de consistencia que deben resolverse para evitar fallos en sistema. En el caso particular de las memorias caché, tanto centralizadas como distribuidas, se utilizan mecanismos conocidos como protocolos de consistencia de caché los cuales se basan en una técnica llamada *Snoopy cache*.

2.5.1. *Snoopy cache*

Snoopy o *Snooping cache* es una técnica utilizada en los sistemas organizados alrededor de un bus, con la cual múltiples CPU escuchan constantemente el bus independientemente de si deben realizar operaciones con la memoria compartida para conocer que acciones están realizando los demás CPUs del sistema y reaccionar apropiadamente [10]. Esta técnica se utiliza para implementar sobre ella distintos protocolos de consistencia de caché.

2.5.2. *Write-through*

Write-through es un protocolo de consistencia de caché basado en la técnica de *Snoopy cache* [10]. Con este protocolo, si un CPU tiene un cierto dato en su caché entonces realizará las acciones indicadas en la Tabla 1 dependiendo de que acciones realice el mismo u otro CPU en el sistema respectivamente. Este protocolo es sencillo de implementar pero se caracteriza por no ser escalable con el número de CPUs.

Tabla 1: Protocolo *write-through*. Tabla recuperada de [3].

Evento de cache	**Acción tomada por una caché en respuesta a su propio CPU**	**Acción tomada por una caché en respuesta a la acción de otro CPU**
Fallo de lectura	Leer dato de memoria y guardar en caché.	Ninguna acción.
Éxito de lectura	Leer dato de caché.	Ninguna acción.
Fallo de escritura	Actualizar memoria y guardar en caché.	Ninguna acción.
Éxito de escritura	Actualizar memoria y caché.	Invalidar entrada de caché.

2.5.3. Protocolo del propietario

El principal problema con *Write-through* consiste en la sobrecarga sobre el bus que implica el tener que actualizar la memoria cada vez que se realiza una escritura en alguna memoria caché [10]. Para resolver este problema se introducen los protocolos de propietario, los cuales se basan en memorias caché que pueden marcar lineas en tres posibles estados: *inválido* cuando la linea no posee datos útiles; *limpio* cuando la linea se encuentra actualizada aunque puede existir en otras caches; y finalmente *sucio* cuando la linea ha sido modificada y estas modificaciones no existen en otras caches.

El protocolo funciona como sigue [10]. Cuando un CPU quiere leer un dato que ningún CPU posee en caché o si existe en estado *limpio* en otras caches, lo obtiene directamente de la memoria. Luego, si un CPU realiza una escritura en un dato que posee en su caché, esta se realiza de forma local aunque se anuncia por el bus (sin actualizar la memoria principal), de forma que otros CPU que posean el mismo dato en estado *limpio* en sus respectivas caches puedan colocarlo en estado *inválido*; consecuentemente, el CPU que realizo la escritura marca dicho dato como *sucio*. En este momento, si otro CPU desea leer dicho dato, la solicitud de lectura será respondida por el procesador que posee la copia *sucia* en lugar de la memoria compartida. El procesador que respondió a la lectura marcará su copia del dato como *inválida*. Los datos solo son escritos a la memoria compartida cuando el CPU que posea la única copia *sucia* de los mismos debe sustituirlos por otros datos.

2.5.4. Protocolo de invalidación

Los protocolos de consistencia basados en *snoopy cache* funcionan en los sistemas multiprocesadores pero son inaplicables en los sistemas que utilizan DSM [10]. En estos sistemas, particularmente en los que utilizan DSM basada en páginas, se hace uso de los llamados protocolos de invalidación de caché [10].

En un protocolo de invalidación de caché una página puede encontrarse en algún momento dado en uno de dos estados posibles, R (*lectura*) y W (*lectura-escritura*), además de que cada página posee un proceso identificado como su dueño, rol que puede cambiar en tiempo de ejecución y se identifica como el último proceso que realizó una escritura cada página. Para el caso de un proceso que necesita leer una página, el protocolo consiste en identificar que proceso es el dueño actual de dicha página y solicitar una copia en caso de que esta se encuentre en estado R, o primero solicitar el paso de la misma a estado R si se encuentra en estado W y luego solicitar la copia.

En el caso de que el proceso desee escribir a la página el protocolo consiste en identificar el dueño de la misma, solicitar su invalidación (que se realiza mediante un *broadcast*) y luego solicitar la propiedad de la misma , finalmente colocando dicha página en estado W y realizando la escritura. Este protocolo garantiza que solo una copia de cada página puede encontrarse en estado W en un momento dado, evitando así la ocurrencia de inconsistencias [10].

Conclusiones

En este documento se realizó un estudio de las distintas estrategias de manejo de memoria utilizadas en los sistemas operativos centralizados y distribuidos. La memoria en una computadora se organiza en una jerarquía centrada en la cercanía de dicha memoria al procesador, siendo las memorias mas cercanas al CPU más rápidas pero más pequeñas y costosas, como por ejemplo los registros y la memoria caché. Por otra parte, las memorias más lejanas al CPU poseen mayor capacidad de almacenamiento dado que son más económicas por bit, aunque son órdenes de magnitud más lentas que las memorias más cercanas al procesador.

Los sistemas centralizados suelen utilizar estrategias de memoria virtual para poder facilitar la ejecución de múltiples procesos concurrentes obteniendo ventajas como la posibilidad de proteger los espacios de memoria de los procesos de accesos no autorizados por parte de otros procesos, y la posibilidad de mover procesos al almacenamiento secundario para poder ejecutar más procesos de los que caben físicamente en memoria. Otras estrategias como la segmentación, la cual representa la memoria como una estructura bidimensional, también son utilizadas.

En los sistemas distribuidos se utiliza un mecanismo que intenta simular el funcionamiento de una memoria compartida global utilizando las memorias privadas locales de los distintos nodos del sistema. Este mecanismo se conoce como memoria compartida distribuida. En estos sistemas se suelen utilizar estrategias de memoria caché para minimizar las transferencias de datos. Sin embargo, esto trae con sigo la posibilidad de que distintas copias

de un mismo dato puedan existir simultáneamente, siendo necesario entonces el garantizar la consistencia de los datos para evitar que procesos específicos trabajen con copias que han sido modificadas independientemente por otros procesos. Esto se logra utilizando distintos algoritmos de consistencia de caché.

Referencias

[1] C. Hamacher, Z. Vranesic y S. Zaky, *Organización de Computadores*, 5^a Edición, McGraw Hill, 2003.

[2] M. Mano y C. Kime, *Fundamentos de Diseño Lógico y de Computadoras*, 3^a Edición, Pearson, 2005.

[3] A. Tanenbaum, *Modern Operating Systems*, 3^a Edición, Pearson, 2008.

[4] A. Tanenbaum y M. Van Steen, *Sistemas Distribuidos: Principios y Paradigmas*, 2^a Edición, Pearson, 2007.

[5] J. Torres, *Conceptos de Sistemas Operativos: Teoría y Práctica*, 1^a Edición, Trillas, 2001.

[6] A. Tanenbaum y A. Woodhull, *Operating Systems: Design and Implementation*, 3^a Edición, Pearson, 2006.

[7] A. Silberschatz, P. Galvin y G. Gagne, *Fundamentos de Sistemas Operativos*, 7^a Edición, McGraw Hill, 2006.

[8] D. Knuth, *The art of computer programming, volume 1: fundamental algorithms*, Addison Wesley Longman Publishing Co., 1997.

[9] A. Tanenbaum, *Organización de Computadoras: Un Enfoque Estructurado*, 4^a Edición, Prentice Hall, 2000.

[10] A. Tanenbaum, *Distributed Operating Systems*, 1^a Edición, Pearson, 1994.

[11] A. Goscinski, *Distributed Operating Systems: The Logical Design*, 1^a Edición, Addison-Wesly Publishing Company, 1991.

[12] G. Colouris, J. Dollimore y T. Kindberg, *Distributed Systems: Concepts and Design*, 2^a Edición, Addison-Wesley, 1994.