Tema 1

Administración de la memoria

La gestión de la memoria es un aspecto fundamental de los sistemas operativos, ya que para que los procesos puedan ejecutarse deben residir en ella. En este capítulo se estudiarán los requisitos de gestión de memoria, así como diversos esquemas de asignación que exigen que la imagen del proceso esté cargada en ella completamente.

1.1. Jerarquía del almacenamiento

Un sistema de computación necesita disponer de un sistema de almacenamiento eficiente. El sistema ideal sería el que tuviera una gran capacidad para poder ejecutar la mayor cantidad de programas posible, un tiempo de acceso bajo para obtener un mejor rendimiento, y un coste razonable. Sin embargo, no existe ningún sistema de almacenamiento de información que cumpla estos tres requisitos simultáneamente. Normalmente se establecen las siguientes relaciones:

- A menor tiempo de acceso, mayor coste por bit.
- A mayor capacidad, menor coste por bit.
- A mayor capacidad, mayor tiempo de acceso.

La solución al problema consiste en no emplear un único componente de memoria, sino una **jerarquía de almacenamiento**, como la que aparece en la figura 1.1.

Los distintos tipos de memoria que se muestran pueden ser volátiles o no volátiles según si la información almacenada se mantiene de forma permanente o no. Del mismo modo, se pueden clasificar en sistemas de almacenamiento externos o internos, dependiendo de si encuentran o no en dispositivos externos al sistema.

En el nivel superior se encuentra la memoria más rápida, de menor capacidad y mayor coste, los registros del procesador; en contraposición con el nivel más bajo en el que aparece la memoria más lenta, de mayor capacidad y más económica. La **memoria principal** es el elemento principal de almacenamiento interno en el sistema de computación. Con objeto de mejorar el rendimiento se suele contar con un nivel de almacenamiento más rápido y de menor capacidad, que se conoce como **memoria caché**. Ésta se utiliza como almacenamiento intermedio de los datos usados recientemente, para que un posterior uso no implique acceder a memoria principal.

Los dispositivos de almacenamiento masivo externos, tales como los discos, cintas magnéticas y los discos ópticos constituyen un sistema de almacenamiento no volátil de mucha mayor capacidad que la memoria principal. Esta memoria externa no volátil se conoce como **memoria auxiliar** o **secundaria**. Los discos también se suelen emplear para proporcionar una extensión a la memoria principal, conocida como **memoria virtual**, que se discutirá en el capítulo siguiente.

En la figura 1.1 aparece también otro nivel de almacenamiento que es la **caché de disco**, cuyo funcionamiento es similar a la situada entre los registros y la memoria principal.

Podemos decir que a medida que bajamos en la jerarquía se cumplen las siguientes condiciones:

1. Disminuye el coste por bit.
2. Aumenta la capacidad.
3. Aumenta el tiempo de acceso.
4. Disminuye la frecuencia de acceso por parte del procesador.

Este último punto, la disminución de la frecuencia de acceso, es la clave para el éxito de esta jerarquía. La base para la validez de esta afirmación viene dada por el principio conocido como **cercanía de referencias** o **principio de localidad**. En él se establece que durante la ejecución de un proceso, las referencias a memoria que hace el procesador (instrucciones y datos), tienden a estar agrupadas. Es decir, en períodos largos de tiempo los grupos de localizaciones a los que se hace referencia cambian, pero en períodos cortos, el procesador suele trabajar con un conjunto de referencias a memoria que no cambia. Teniendo esto en cuenta, es posible organizar la información a través de la jerarquía, de tal forma que el porcentaje de accesos a cada una de ellas sea sustancialmente más pequeño que aquella a la que precede.

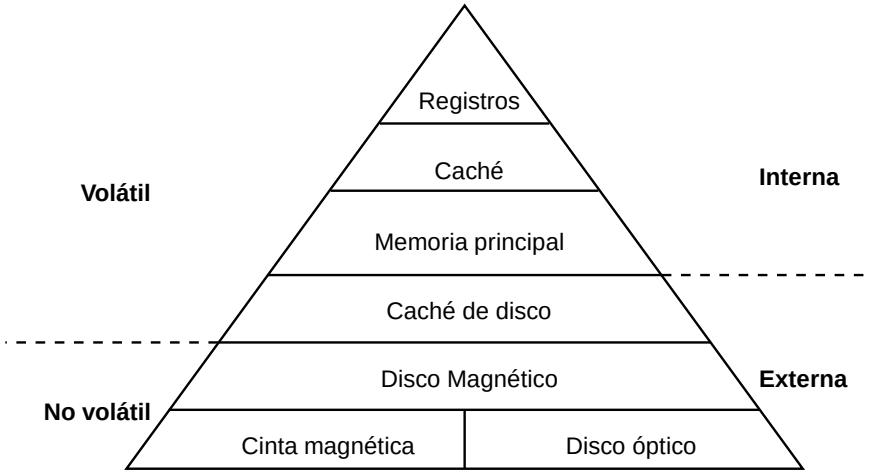

Figura 1.1: Jerarquía del almacenamiento

1.2. Traducción de direcciones

En el tema 3 hicimos referencia a que un programa se convierte en proceso cuando se carga en memoria. Para ello, el programa debe pasar previamente por una serie de fases como son la compilación, el enlazado y por último su carga en la memoria principal.

Un proceso en memoria consta de instrucciones más datos. Las instrucciones pueden contener referencias a memoria de dos tipos:

- Direcciones de datos.
- Direcciones de otras instrucciones.

Las direcciones de memoria se pueden expresar de distintas formas, así podemos distinguir entre **direcciones lógicas**, **relativas** y **físicas**. Una dirección lógica o **simbólica** es una referencia a una localización de memoria independiente de la asignación actual de datos a la memoria; antes de acceder a la memoria habrá que realizar una traducción de tales direcciones a direcciones físicas. Una dirección relativa es un ejemplo particular de dirección lógica, en la cual la dirección se expresa como una localización relativa a algún punto conocido, usualmente el principio del programa. Una dirección física o absoluta, es una localización en memoria principal. La figura 1.2 muestra los posibles direccionamientos.

En las diferentes etapas por las que pasa el programa antes de poder ser ejecutado, las direcciones de memoria se pueden representar de distintas formas. Sin embargo, tenemos que tener en cuenta que en el momento de la ejecución siempre ha de hacerse referencia a una dirección física, por lo que si inicialmente se utilizan direcciones lógicas éstas tendrán que ser traducidas a direcciones físicas en algún momento. Esta **traducción** o **ligadura de direcciones** se puede producir:

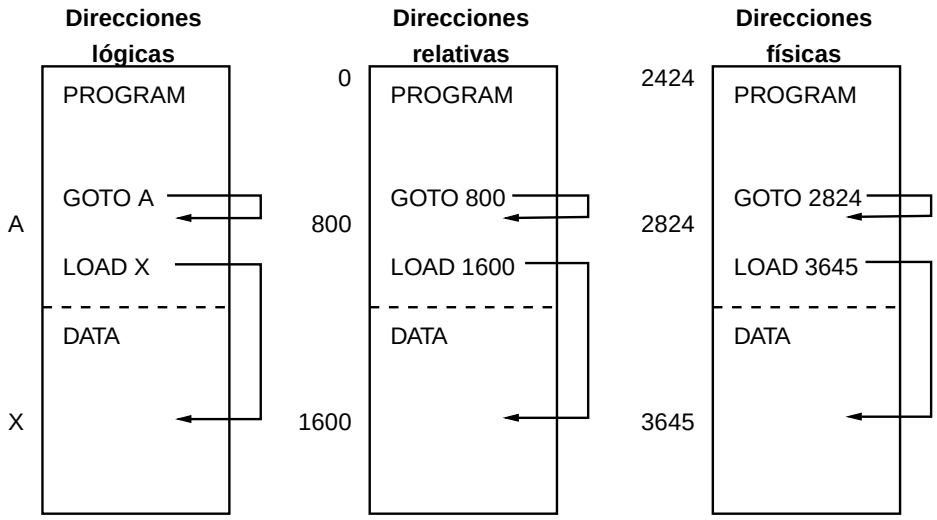

Figura 1.2: Tipos de direccionamientos

- En tiempo de programación o compilación.
- En tiempo de carga.
- En tiempo de ejecución.

Si en el momento de hacer el programa, el programador proporciona direcciones absolutas o si el compilador genera este tipo de direcciones, el módulo de carga que se le pasa al cargador contendrá direcciones absolutas, por lo que tendrá que ser colocado siempre en la misma zona de memoria. Esto presenta dos inconvenientes; por un lado, no permite cambiar el proceso de ubicación durante su ejecución; por otro, es necesario conocer de antemano en qué zona de la memoria se va a situar el proceso. Esto requerirá tener conocimiento de la técnica de gestión de memoria que se emplea, con objeto de saber qué huecos libres existen en ella.

La desventaja de generar direcciones absolutas antes de la carga es que el módulo de carga resultante sólo puede ser colocado en una región específica de la memoria principal. Cuando la memoria principal es compartida por muchos procesos esta forma de trabajo es poco flexible, por tanto es mejor tener un módulo de carga que pueda ser colocado en cualquier parte de la memoria.

Para conseguir esto, el compilador no debe producir direcciones absolutas sino direcciones que sean relativas a algún punto conocido, tal como el comienzo del programa. Al principio del módulo de carga se le asigna la dirección relativa 0, y todas las otras referencias a memoria dentro del módulo son expresadas con relación a este comienzo.

Teniendo todas las referencias a memoria expresadas de forma relativa, es fácil para el cargador colocar el módulo en la dirección deseada. Si el módulo va a ser colocado empezando en la dirección x, dirección almacenada en un registro base,

el cargador sólo tendrá que añadir x a cada referencia a memoria a medida que se carga el módulo en memoria.

Con este tipo de traducción, el problema que se presenta es que si se saca el proceso de memoria principal, cuando vuelva a ella deberá ser colocado en la misma posición donde estaba inicialmente. Otro inconveniente es que no se podrá cambiar de ubicación al proceso durante su ejecución.

La última posibilidad es retrasar el cálculo de las direcciones absolutas hasta que se necesiten, es decir, en tiempo de ejecución. En este caso, el módulo de carga es situado en memoria principal con todas las referencias a memoria en forma relativa, y hasta que no se ejecute una instrucción no se calculan las direcciones absolutas. Para que esto no degrade el rendimiento del sistema, se debe proporcionar un mecanismo hardware especial que describiremos un poco más adelante.

Este cálculo dinámico de direcciones proporciona una flexibilidad completa. Un programa puede ser cargado en cualquier zona de la memoria principal, el proceso se podrá intercambiar sin problemas y, si es necesario, podrá cambiar de ubicación en memoria principal.

La figura 1.3 muestra la forma en que se hace normalmente la traducción de direcciones. Cuando un proceso pasa al estado de ejecución, un registro especial del procesador, llamado a veces registro base, se carga con la dirección de comienzo del proceso en memoria principal. También existe un registro límite que indica la última dirección correspondiente al proceso. Durante la ejecución del proceso nos encontramos con direcciones relativas que tienen que ser traducidas a direcciones absolutas. Para ello la dirección relativa es manipulada en dos pasos por el procesador. En primer lugar, se añade el valor del registro base a la dirección relativa para producir la dirección absoluta. En segundo lugar, la dirección resultante se compara con el valor del registro límite. Si la dirección está dentro de los límites, se puede proceder a la ejecución de la instrucción. Si no es así, se genera una excepción, para que el sistema operativo trate el error. En un entorno de multiprogramación estos registros deberán guardarse en el BCP cuando se hace un cambio de proceso.

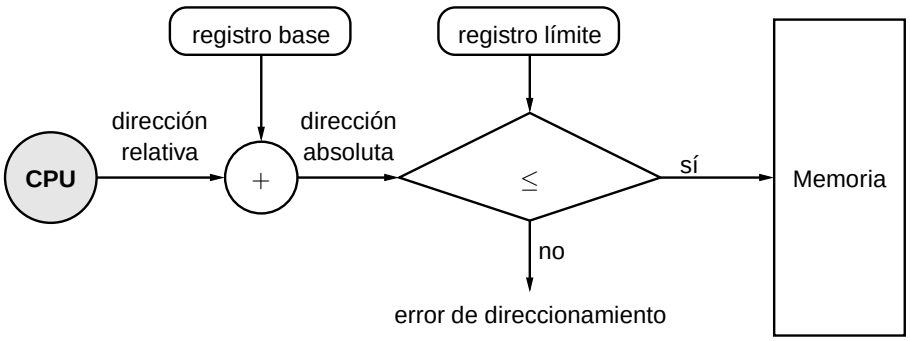

Figura 1.3: Soporte hardware para la traducción de direcciones relativas

El esquema de la figura 1.3 permite que los programas sean intercambiados du-

rante el curso de su ejecución. También proporciona un método de protección: la imagen de cada proceso se encuentra aislada por el contenido de los registros base y límite, estando a salvo del acceso por parte de otros procesos.

1.3. Funciones del administrador de la memoria

Históricamente la memoria principal o primaria ha sido un recurso costoso, por lo que los diseñadores de sistemas operativos han intentado obtener el máximo rendimiento de este recurso. En la actualidad la memoria principal es menos costosa pero sigue siendo importante administrarla adecuadamente porque esto influirá en gran medida en el rendimiento global del sistema.

La parte del sistema operativo que se encarga del manejo de la memoria se denomina **administrador de la memoria**. Entre las labores que debe realizar están:

- Controlar qué partes de la memoria están en uso y cuáles están disponibles.
- Decidir dónde situar los procesos en memoria, y controlar en qué zona de la memoria residen en cada momento.
- Asignar memoria a los procesos cuando la necesiten y retirársela cuando terminen.
- Controlar el intercambio de procesos entre memoria principal y secundaria.

Como consecuencia de estas funciones el administrador de la memoria debe ser capaz de satisfacer tres requisitos: **reubicación**, protección y compartición.

Por reubicación entendemos la posibilidad de que un proceso a lo largo de su vida pueda cambiar de localización dentro de la memoria. En sistemas de multiprogramación esto suele ser necesario por ejemplo, si necesitamos intercambiar el proceso, cuando éste vuelva a la memoria principal puede no ser posible cargarlo en el mismo lugar donde estaba inicialmente; aún en el caso de que no necesitemos intercambiar el proceso, muchos sistemas de administración de la memoria necesitan cambiar los procesos de localización para hacer así un mejor uso de la memoria.

El administrador de la memoria debe proporcionar un sistema de protección adecuado. Tanto en sistemas de monoprogramación como en los de multiprogramación, un proceso de usuario no debe ser capaz de acceder a la zona de memoria donde reside el sistema operativo; tampoco deberá acceder de una forma indiscriminada a la memoria ocupada por otros procesos de usuario.

En los sistemas de multiprogramación nos podemos encontrar con que varios procesos pueden estar ejecutando el mismo código; en estos casos, resulta beneficioso para el sistema que todos ellos compartan la zona de memoria donde reside el código, por lo que el mecanismo de protección que proporciona el sistema debe ser

lo suficientemente flexible como para permitir la compartición de ciertos bloques de memoria por parte de distintos procesos.

1.4. Esquemas de asignación de la memoria

En este capítulo vamos a analizar distintos esquemas de administración de la memoria. La elección de un esquema de administración de la memoria para un sistema concreto depende de muchos factores, especialmente del diseño hardware de éste.

Podemos hacer una clasificación de los esquemas de administración de la memoria:

Asignación contigua Exigen que el proceso se cargue en la memoria ocupando un espacio de direcciones contiguo. Ejemplos de esquemas de asignación de este tipo son:

- Máquina desnuda: No existe sistema operativo, por tanto el proceso de usuario puede ocupar toda la memoria.
- Sistemas de monoprogramación: En memoria principal residen el sistema operativo y un programa de usuario.
- Sistemas de multiprogramación con particiones: La memoria se divide en una serie de fracciones o particiones, residiendo en cada una de ellas un proceso.

Asignación no contigua En este caso no se exige que el proceso ocupe un espacio de direcciones contiguo. Se consideran dos esquemas de este tipo: la **paginación** y la **segmentación**. Debemos tener en cuenta que estos esquemas se utilizan en los sistemas de memoria virtual que se estudiarán en la asignatura Sistmeas Opertivos II.

1.5. Sistemas de monoprogramación

Desde el punto de vista del manejo de la memoria, los sistemas de monoprogramación son mucho más simples que los sistemas de multiprogramación, ya que la memoria está dividida simplemente en dos áreas contiguas. Una de ellas está asignada permanentemente a la parte residente del sistema operativo, mientras que la memoria restante se asigna a los procesos transitorios. En esta área se pueden cargar tanto procesos de usuario como las partes no residentes del sistema operativo. Esta forma de gestión de la memoria la utilizan habitualmente sistemas operativos tales como MS-DOS.

Con el fin de proporcionar un área contigua de memoria libre para los procesos transitorios, el sistema operativo se encuentra generalmente ubicado en un extremo

de la memoria. La elección de un extremo particular de la memoria, la parte superior o la inferior, suele venir determinada por la ubicación del vector de interrupciones. El sistema operativo se coloca habitualmente en el mismo extremo de la memoria donde reside el vector de interrupciones. A veces, partes del sistema operativo se colocan en el extremo opuesto de la memoria dejando así un área contigua grande y única de memoria libre en medio (figura 1.4).

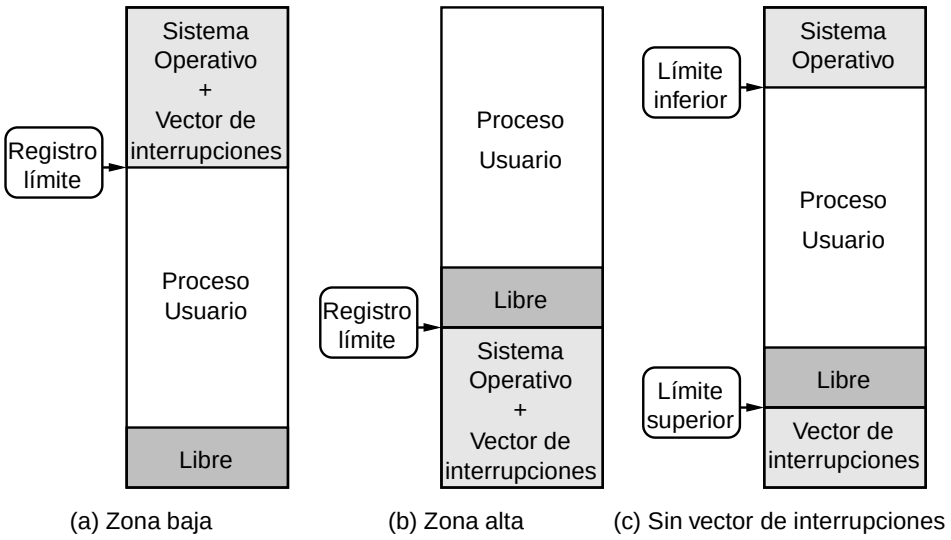

Figura 1.4: Organización de la memoria en monoprogramación

Los sistemas de monoprogramación no suelen proporcionar protección de la zona de memoria donde residen los procesos de usuario, ya que en cada momento sólo permite que haya un único proceso residente en memoria. Sin embargo, es deseable proteger el código del sistema operativo para que no sea deteriorado por un proceso transitorio en ejecución. La protección se puede implementar mediante un registro de la CPU, que se denomina **registro límite**, y que contiene la dirección de memoria a partir de la cual puede acceder el proceso de usuario. Cada vez que éste hace referencia a una dirección de memoria, se comprueba si le está permitido o no, haciendo uso del registro límite, como se puede ver en la figura 1.5. Para que este método sea eficiente, la modificación del registro límite debe ser una operación privilegiada, de este modo los procesos de usuario no podrán modificarlo. Este método requiere, por tanto, la distinción entre dos modos de ejecución, el modo usuario y el modo supervisor.

Algunos ejemplos de sistemas operativos de monoprogramación son el IBM 1130 y el sistema del ordenador Spectrum que cargan el sistema operativo en memoria baja, y el HP 2116B que carga el sistema operativo en memoria alta.

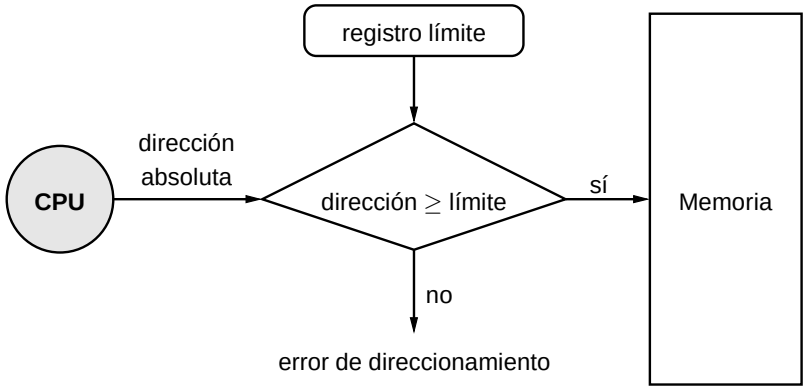

Figura 1.5: Protección de la memoria en sistemas de monoprogramación

1.6. Multiprogramación con particiones fijas

En sistemas de multiprogramación la zona de memoria disponible para los procesos transitorios debe ser compartida por varios procesos, por lo tanto, se necesita un esquema de manejo de la memoria más complejo.

El esquema más simple de manejo de la memoria para sistemas de multiprogramación consiste en dividir la memoria disponible para los procesos transitorios en un número fijo de particiones, cada una de las cuales puede ser asignada a un proceso.

La división de la memoria en particiones se realiza durante el proceso de generación del sistema, por lo que el número y tamaño de éstas permanece fijo a partir de este momento.

A continuación veremos los aspectos más importantes de este esquema de asignación de memoria.

1.6.1. Selección del tamaño de las particiones

En este sistema de administración de la memoria, el número y tamaño de las particiones es fijo, por lo que habrá que determinarlo de antemano. Hay que tener en cuenta que el número de particiones va a determinar el grado de multiprogramación máximo que podemos alcanzar, ya que en una partición sólo se puede cargar un proceso. El tamaño de las particiones también es importante ya que si tenemos un proceso con un tamaño superior al de todas las particiones, no se podrá ejecutar, por tanto, tendremos que hacer estimaciones acerca de cuál puede ser el tamaño máximo de un proceso y del tamaño más usual de éstos. Hay que tener en cuenta también que si el tamaño de un proceso es menor que el de la partición, toda la memoria que sobra se desperdicia ya que no puede ser utilizada por otro proceso, esto es lo que se conoce como **fragmentación interna**.

La figura 1.6 muestra las dos alternativas posibles con respecto al tamaño de las

particiones. Una posibilidad es hacer uso de particiones de igual tamaño y la otra tener particiones de tamaños diferentes. El uso de particiones de tamaños diferentes proporciona una mayor flexibilidad al esquema de particiones fijas.

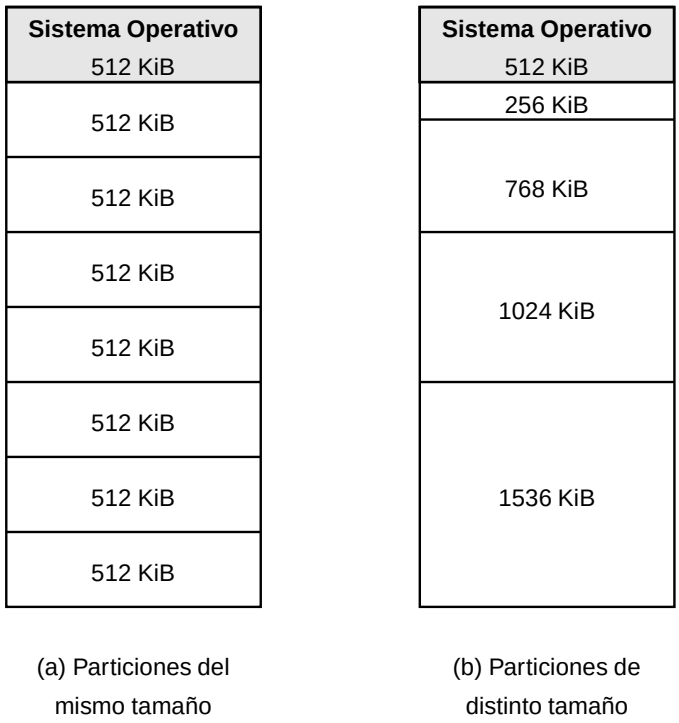

Figura 1.6: Diseño de particiones fijas en una memoria de 4 MiB

1.6.2. Algoritmos de colocación

Si se están usando particiones de igual tamaño, la colocación de los procesos en memoria es trivial. Un proceso se podrá colocar en cualquier partición que esté disponible, siempre que el tamaño de ésta sea mayor o igual que el del proceso. Si todas las particiones están ocupadas, se puede intercambiar algún proceso que no esté en estado listo, para dar paso al nuevo proceso.

Cuando tenemos particiones de tamaños diferentes, hay distintas posibilidades a la hora de asignar los procesos a las particiones. Podríamos tener una cola de procesos asignada a cada partición, donde se colocarían los procesos según su tamaño; de este modo, cada proceso iría a la cola asociada a aquella partición de menor tamaño donde éste cupiera. Cuando una partición queda libre se elige un proceso de la cola asociada para ocuparla. La ventaja de este enfoque es que se minimiza la cantidad de memoria que se desperdicia dentro de cada partición, es decir, se reduce la fragmentación interna. Sin embargo, desde el punto de vista del sistema tiene el inconveniente de que puede haber particiones grandes vacías mientras que tenemos procesos esperando en colas asociadas a particiones pequeñas.

Una organización alternativa consiste en tener una única cola de procesos como en la figura 1.7(b). En este caso, cuando una partición queda libre, es necesario disponer de una estrategia que determine qué proceso de la cola se va a cargar en la partición. Una posibilidad es emplear el algoritmo del **primer ajuste**, que selecciona el primer trabajo de la cola que quepa en la partición. Como no es deseable desperdiciar una partición grande con un trabajo pequeño, una estrategia diferente consiste en buscar en toda la cola de espera cada vez que una partición queda libre y tomar el trabajo más grande que quepa en ella. Este segundo algoritmo discrimina a los trabajos pequeños frente a los grandes, y se conoce con el nombre de **mejor ajuste**.

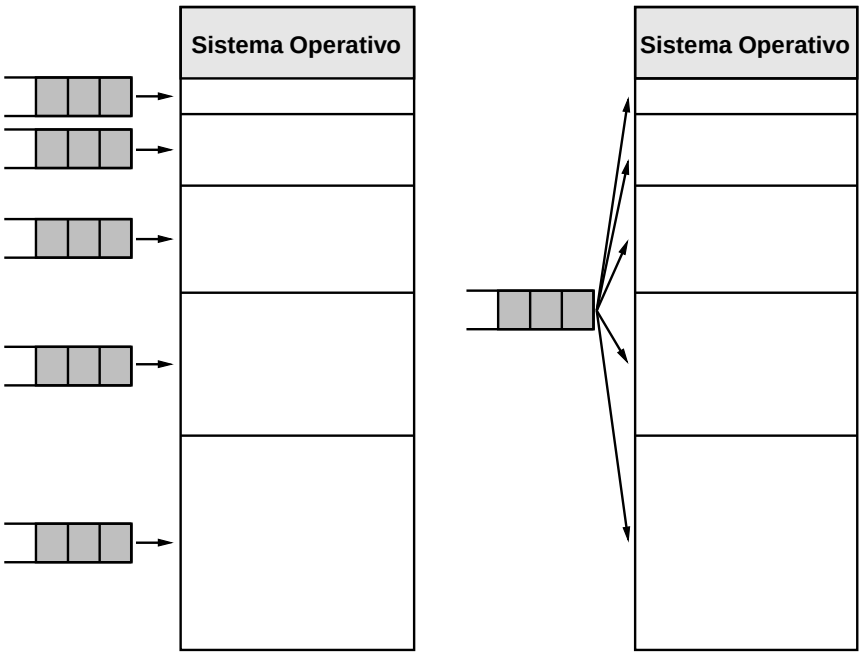

Figura 1.7: Alternativas de planificación en multiprogramación con particiones fijas

La labor de asignar los procesos a las particiones siguiendo una determinada estrategia la realizará en el sistema el planificador a largo plazo del que se habló en el tema 4.

1.6.3. Elementos de control

En un sistema de este tipo se necesita saber en todo momento qué particiones están ocupadas y cuáles están libres, para ello se utiliza una **tabla de particiones**. En ella se describe cada partición del sistema y su estado actual. Un ejemplo de tabla de particiones es la que aparece en la figura 1.8, en ella, cada partición viene descrita por su dirección inicial (base) y su tamaño; también se especifica su estado,

es decir, si la partición está o no asignada a un proceso actualmente. Cuando se va a crear un proceso nuevo, el sistema comprueba si existe una partición del tamaño adecuado, buscando en la tabla de particiones; si es así, marca la partición como asignada y carga el proceso en dicha partición.

Partición	Base	Tamaño	Estado
0	0 KiB	100 KiB	ASIGNADA
1	100 KiB	300 KiB	LIBRE
2	400 KiB	100 KiB	ASIGNADA
3	500 KiB	250 KiB	ASIGNADA
4	750 KiB	150 KiB	ASIGNADA
5	900 KiB	100 KiB	LIBRE

Figura 1.8: Tabla de particiones para un sistema de particiones fijas

1.6.4. Protección

En los sistemas de multiprogramación debemos proteger el código y los datos de cada proceso frente al acceso de los demás. La protección se suele implementar mediante varios registros límite. Se pueden utilizar dos registros para delimitar los extremos superior e inferior de una partición, o bien un solo registro y la longitud de ésta. La figura 1.9 muestra ambos esquemas de protección.

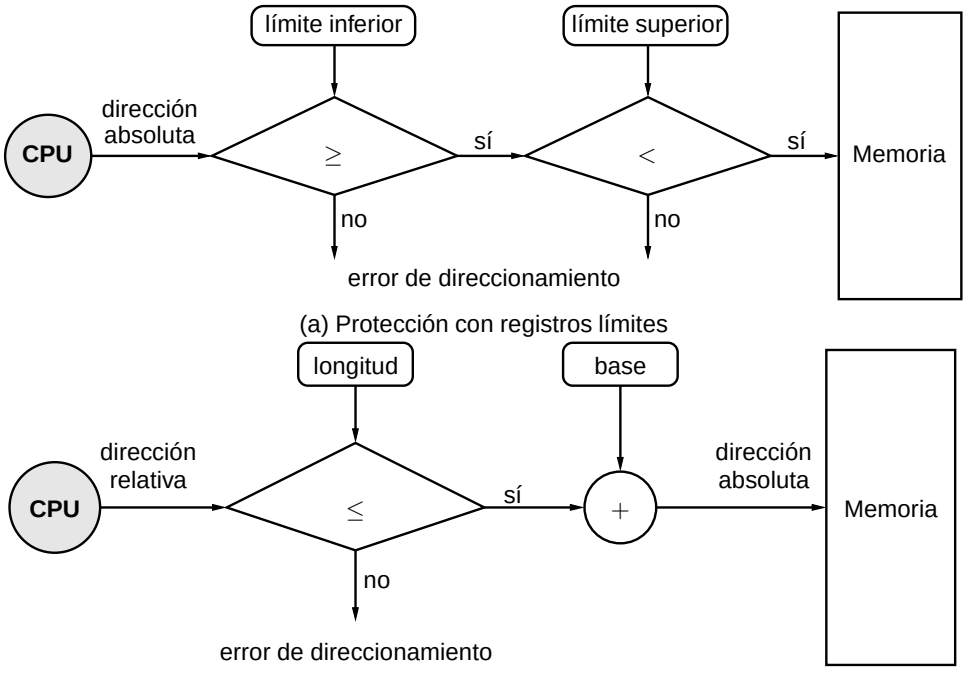

Figura 1.9: Esquemas de protección en sistemas con particiones fijas

Cada vez que un proceso desea acceder a una dirección de memoria se comprueba si está dentro de los límites permitidos para este proceso; si es así se le dejará acceder a ella, si no lo está se producirá un error.

1.6.5. Inconvenientes

El esquema de particiones fijas es muy simple, pero presenta algunas desventajas:

- El número de particiones especificadas en el momento de la generación del sistema limita el número de procesos activos en el sistema, es decir, el grado de multiprogramación.
- Los trabajos pequeños que no ocupen el espacio completo de una partición provocan fragmentación interna.
- Dado que el tamaño de las particiones se establece en el momento de la generación del sistema, si el tamaño de un proceso es mayor al de cualquiera de las particiones definidas no se podrá ejecutar en el sistema, a menos que se vuelva a redefinir el tamaño de las particiones.

El uso de las particiones fijas es casi desconocido hoy día. Un ejemplo de un sistema operativo que usó esta técnica fue un sistema de IBM para grandes computadores, el OS/MFT (*Multiprogramming with a Fixed Number of Tasks*).

1.7. Multiprogramación con particiones variables

Para vencer algunas de las dificultades que presentaban las particiones fijas, se desarrolló otro esquema de manejo de la memoria conocido como particiones variables o dinámicas. Un sistema operativo importante que usó esta técnica fue el sistema de IBM para grandes computadores, OS/MVT (*Multiprogramming with a Variable Number of Tasks*).

Cuando se emplean particiones variables, el número y tamaño de éstas varía dinámicamente a lo largo del tiempo, ya que cada proceso ocupa la cantidad de memoria que necesita. En la figura 1.10 puede verse cómo funciona. Inicialmente, sólo tenemos en memoria el sistema operativo (figura 1.10a). Posteriormente se van cargando los procesos 1, 2 y 3 (figuras 1.10b, c y d). Esto deja un hueco que no es lo suficientemente grande como para albergar al proceso 4. En un instante dado, el sistema intercambia el proceso 2 (figura 1.10e), dejando así espacio suficiente para el proceso 4 (figura 1.10f). Termina la ejecución del proceso 1 y abandona la memoria (figura 1.10g). Posteriormente se vuelve a cargar en memoria el proceso 2 que había sido intercambiado (figura 1.10h).

La diferencia principal entre el método de particiones fijas y el de particiones variables es que el número, posición y tamaño de las particiones varían de forma

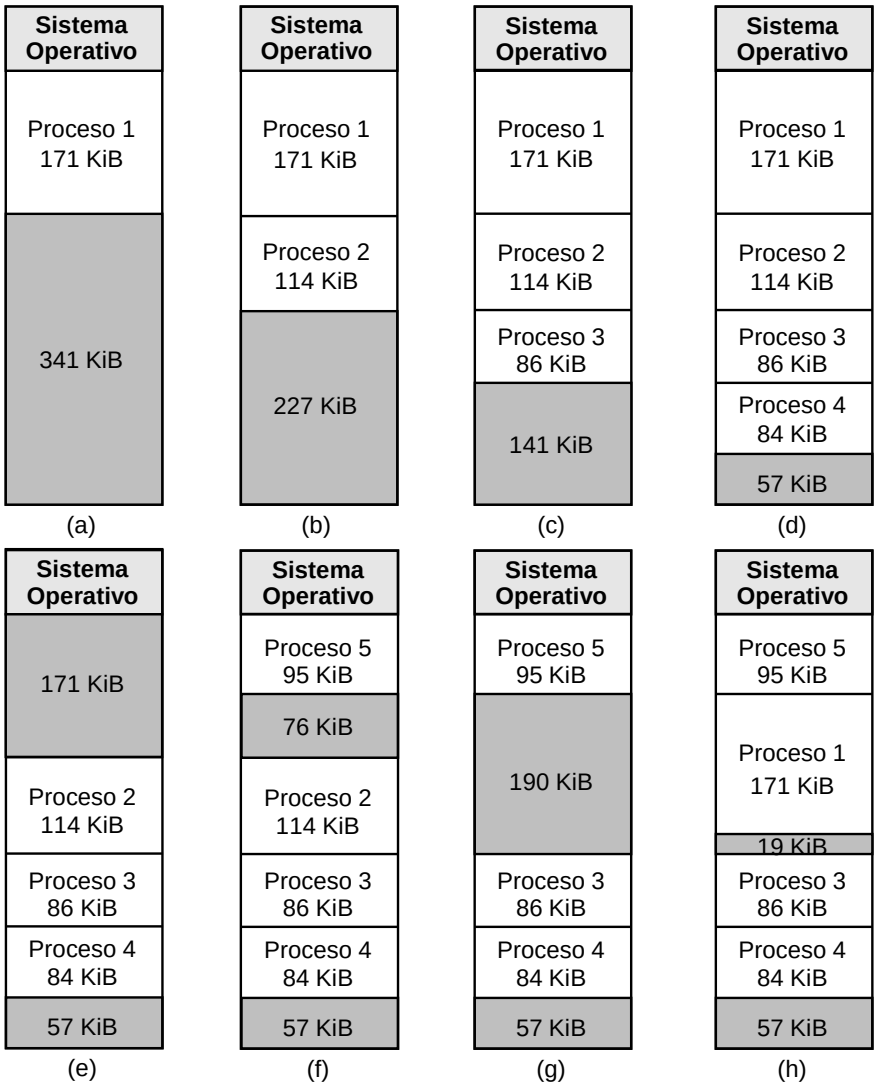

Figura 1.10: Asignación de la memoria con particiones variables

dinámica en el segundo a medida que los procesos llegan o se van, mientras que en el primero son fijos. La flexibilidad de no atarse a un número fijo de particiones mejora la utilización de la memoria, pero a su vez complica la asignación y desasignación de la memoria, así como su control.

El método de las particiones variables, en principio, funciona bien, pero puede conducir a una situación en la que hay una gran cantidad de huecos pequeños en la memoria, que no pueden ser aprovechados para cargar nuevos procesos, esto es lo que se conoce como **fragmentación externa**.

1.7.1. Compactación

Una técnica que permite solucionar la fragmentación externa es la **compactación** o **recolección de basura** (*garbage collection*). Cada cierto tiempo el sistema operativo se encarga de desplazar los procesos cargados en memoria de forma que todo el espacio libre quede formando un solo hueco. Si en la figura 1.10h hacemos compactación nos quedaría un hueco de 256 KiB. El problema que presenta la compactación es que consume tiempo de CPU. La figura 1.11 muestra de forma esquemática la memoria antes y después de realizar la compactación.

¿En qué momentos debería realizar la compactación el sistema operativo? Tenemos tres posibilidades. Podemos realizarla cuando se alcance un determinado nivel de ocupación de la memoria, por ejemplo, el 75 %. La desventaja de ésta es que podemos obligar al sistema a realizar un trabajo extra sin ser necesario porque no haya procesos esperando a entrar en memoria. La segunda posibilidad es hacer compactación de la memoria sólo cuando haya procesos esperando a entrar y los huecos no tengan el tamaño adecuado. En este caso habrá que estar comprobando continuamente si hay trabajos en espera. La tercera posibilidad es hacerla cada cierto tiempo. Si el intervalo de tiempo es pequeño se gastará mucho tiempo en compactación, si se elige demasiado grande podrán agruparse muchos trabajos a la espera y pueden perderse las ventajas de la compactación.

No siempre es posible realizar la compactación, para que funcione requiere que los procesos se puedan ejecutar en su nueva ubicación, por tanto, éstos deben ser reubicables (sección 1.2).

1.7.2. Algoritmos de colocación

Dado que la compactación de la memoria consume tiempo, esto hace que los diseñadores del sistema operativo tengan que ser cuidadosos con la forma de colocar los procesos en memoria. A la hora de cargar o intercambiar un proceso en memoria principal y si hay más de un hueco libre de tamaño suficiente, el sistema operativo deberá decidir cuál de ellos le asigna.

Podemos considerar cuatro algoritmos de colocación, que son: el mejor ajuste, el primer ajuste, el **siguiente ajuste** y el **peor ajuste**.

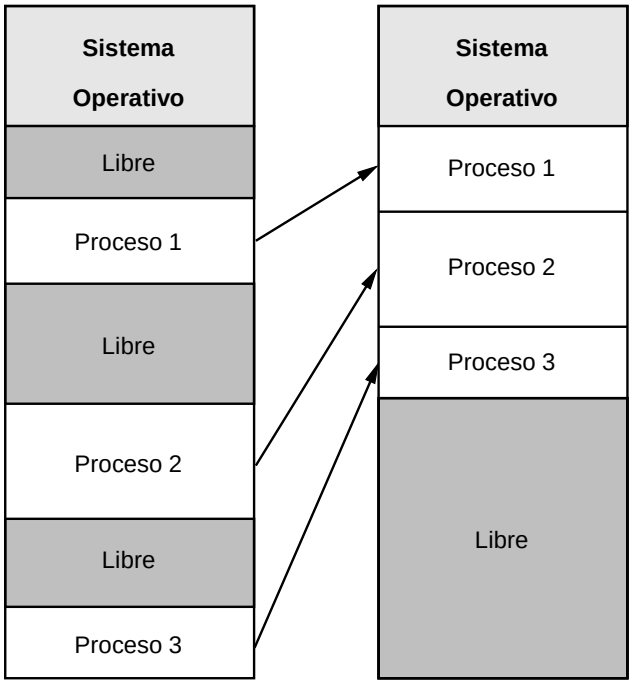

Figura 1.11: Compactación en un sistema con particiones variables

El algoritmo del mejor ajuste elige el bloque de memoria que tiene un tamaño más parecido al del proceso. El primer ajuste empieza a rastrear la memoria desde el principio y elige el primer bloque disponible que sea lo suficientemente grande para alojar al proceso. El siguiente ajuste empieza a rastrear la memoria desde la localización de la última asignación y elige el siguiente bloque disponible que sea lo suficientemente grande. El peor ajuste hace lo contrario del primero, es decir, elige para cada proceso el hueco más grande posible.

La figura 1.12a muestra un ejemplo de configuración de la memoria después de un cierto tiempo, indicándose el último bloque que ha sido asignado. En la figura 1.12b se muestra la diferencia entre los algoritmos del mejor, el peor, el primer y el siguiente ajuste al satisfacer una petición de 16 KiB. El mejor ajuste buscará en la lista completa de bloques libres y hará uso del bloque de 18 KiB, dejando un fragmento de 2 KiB. El primer ajuste da lugar a un fragmento de 6 KiB, el siguiente y el peor ajuste obtienen un fragmento de 20 KiB.

Estudios comparativos de los algoritmos han conducido a las siguientes conclusiones. El algoritmo del primer ajuste no sólo es el más simple sino que a menudo es también el más rápido y el mejor. El algoritmo del siguiente ajuste tiende a producir resultados ligeramente peores que el primer ajuste, ya que con frecuencia, conduce a la asignación de un bloque libre del final de la memoria, que suele ser el bloque más grande, con lo cual al cabo de cierto tiempo, éste se fragmenta. Por tanto, con el siguiente ajuste puede ser necesaria más frecuentemente la compactación. El algoritmo del mejor ajuste, a pesar de su nombre, es usualmente el que da peores resultados. Puesto que busca el bloque más pequeño que se ajusta a la petición, tam-

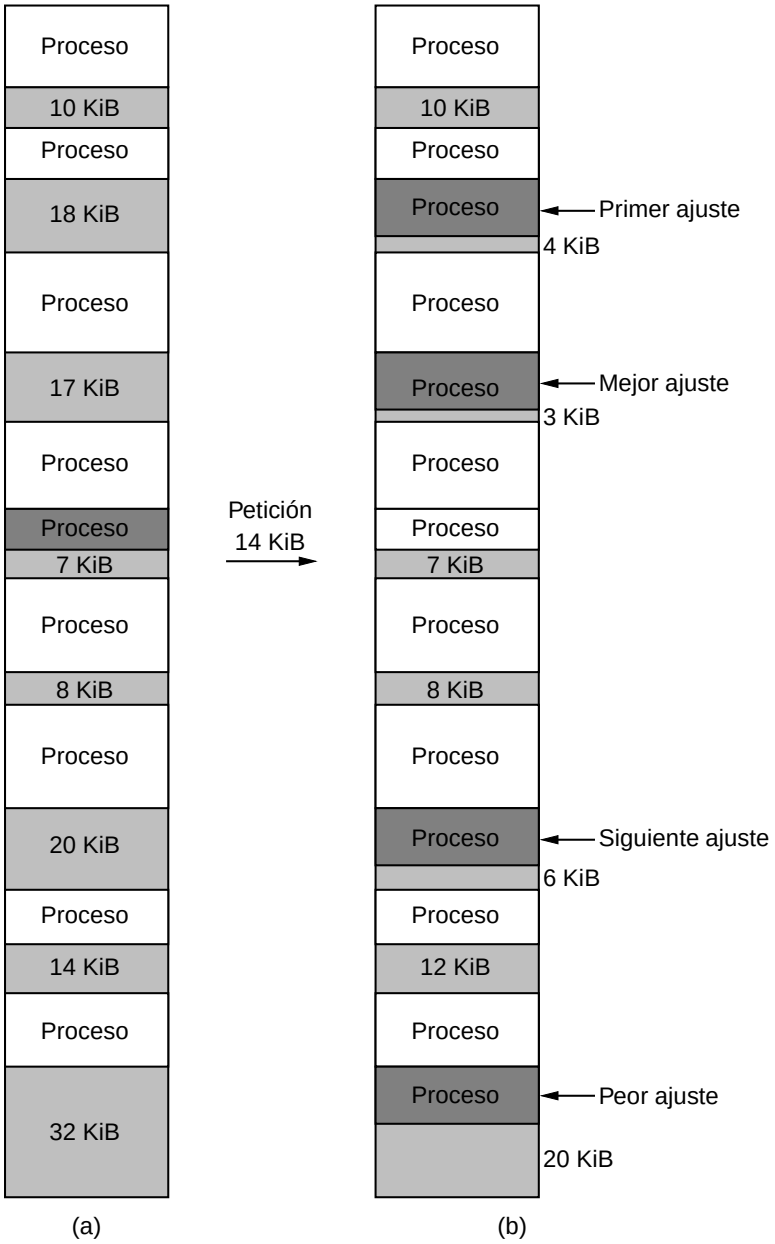

Figura 1.12: Ejemplo de uso de diversos algoritmos de colocación

bién garantiza que el fragmento que se produce será el más pequeño posible. Aunque en cada petición de memoria se gaste la menor cantidad posible de ésta, resulta que la memoria se fragmenta rápidamente en huecos demasiado pequeños para satisfacer otras peticiones. Por tanto, se necesita hacer compactación más frecuentemente que con los otros algoritmos. Por esta razón, se puede utilizar también el algoritmo del peor ajuste, para así dejar huecos grandes que puedan ser aprovechados por otros procesos.

1.7.3. Elementos de control

Vamos a considerar dos métodos que pueden utilizarse para llevar el control del uso de la memoria en sistemas de particiones variables, estos son los mapas de bits y las listas enlazadas, que pasaremos a analizar a continuación.

1.7.3.1. Mapas de bits

Cuando se emplean mapas de bits, la memoria se divide en unidades de asignación; el tamaño de estas unidades puede variar desde unas cuantas palabras hasta varios KiB. En correspondencia con cada unidad de asignación hay un bit en el mapa de bits, que está a 0 si la unidad está libre y a 1 si está ocupada (o viceversa). En la figura 1.13 se representa una parte de la memoria y el mapa de bits correspondiente.

El tamaño de la unidad de asignación es un aspecto importante en el diseño. Cuanto menor es la unidad de asignación, tanto mayor es el mapa de bits (más memoria ocupa). Si la unidad de asignación se escoge grande, el mapa de bits será pequeño, pero puede desperdiciarse memoria si el tamaño del proceso no es un múltiplo entero de la unidad de asignación.

Un mapa de bits es una manera sencilla de llevar el control de la situación de la memoria en una cantidad de memoria fija, ya que el tamaño del mapa de bits depende únicamente del tamaño de la memoria principal y del tamaño de la unidad de asignación. El problema principal que tiene este método es que la búsqueda en un mapa e bits de un conjunto de bits con una longitud determinada es una operación lenta.

1.7.3.2. Listas enlazadas

Otra manera de llevar el control de la memoria es mediante la conservación de una lista enlazada de los segmentos de memoria asignados y libres, donde cada segmento representa a un proceso o a un hueco entre dos procesos. La figura 1.14 se representa una zona de la memoria y su lista enlazada. Cada elemento de la lista especifica un hueco (H) o un proceso (P), la dirección de comienzo, su longitud y un puntero al siguiente elemento de la lista.

La lista enlazada del ejemplo está ordenada por direcciones. Esto presenta la

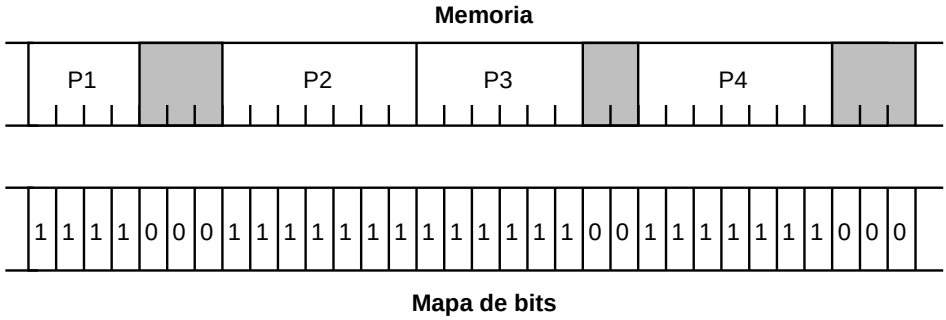

Figura 1.13: Mapa de bits de una zona de memoria con 5 procesos y 3 huecos

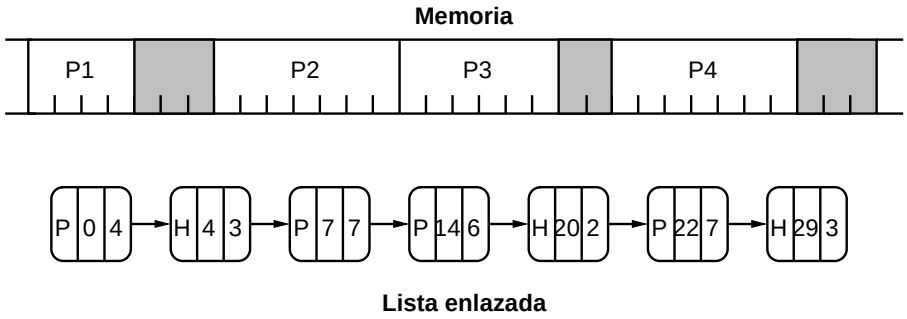

Figura 1.14: Lista enlazada de una zona de memoria con 5 procesos y 3 huecos

ventaja de que la actualización de la lista es inmediata cuando un proceso termina o se intercambia. Un proceso normalmente tiene dos vecinos (excepto cuando está en la parte alta o baja de la memoria), éstos pueden ser procesos o huecos, dando lugar a cuatro combinaciones posibles (figura 1.15). En algunos casos, la actualización de la lista consistirá simplemente en sustituir el tipo de segmento (proceso por hueco) (figura 1.15d. En otros, cuando hubiera un hueco adyacente al proceso, dos entradas se convertirán en una sola (figura 1.15b y c). Si el proceso estaba rodeado por dos huecos, tres entradas se convertirán en una (1.15a). Dado que la entrada correspondiente al proceso en la tabla de procesos apunta normalmente a la entrada de la lista correspondiente al éste, podría ser más conveniente mantenerla como una lista doblemente enlazada, ya que esta estructura hace más fácil encontrar la entrada previa y ver si es posible la fusión de dos huecos adyacentes.

Otra posibilidad consiste en utilizar listas separadas para llevar el control del espacio ocupado y el de los huecos libres; esta última puede mantenerse ordenada por tamaño, de esta forma es más fácil y rápido la búsqueda de un hueco de un tamaño adecuado.

Cada lista se puede ordenar por el criterio que se estime oportuno adaptándose al algoritmo de gestión de memoria que se haya implementado. Por ejemplo, si se

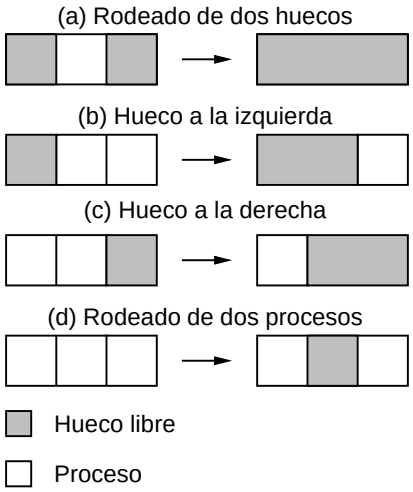

Figura 1.15: Combinaciones posibles cuando un proceso termina

está usando el algoritmo del mejor ajuste, podríamos mantener la lista de huecos ordenada de menor a mayor tamaño; así, cuando se encuentra un hueco que se ajusta al tamaño del proceso se sabe que éste es el más pequeño posible y en consecuencia, el mejor ajuste. Con una lista de huecos ordenada por tamaño el primer y el mejor ajuste son igualmente veloces.

Cuando los huecos se conservan en una lista separada de los procesos es posible una pequeña optimización. En vez de tener un conjunto aparte de estructuras de datos para conservar la lista de huecos, se pueden usar los mismos huecos, así no se necesita memoria adicional para mantener la lista. Sólo necesitaríamos un puntero al primer hueco libre. Cada hueco contendría como entrada su tamaño, un puntero al siguiente hueco y otro al anterior, con objeto de facilitar las fusiones de huecos.

1.8. El sistema compañero

Los sistemas de multiprogramación con particiones presentan varios inconvenientes. Por un lado, en un esquema de particiones fijas se limita el grado de multiprogramación del sistema y se desperdicia espacio mediante la fragmentación interna. Por otro, el esquema de particiones variables presenta una sobrecarga en el sistema con la realización de la compactación para resolver la fragmentación externa, y un mantenimiento más complejo. Un compromiso entre ambos esquemas es el **sistema compañero**.

En este sistema, no se establece un número fijo de particiones, pero se limita el tamaño de los bloques de memoria que se pueden solicitar para alojar a un proceso. Sólo es posible asignar bloques de memoria de tamaño 2^k, donde $I \leq k \leq S$, donde 2^I es el tamaño más pequeño que se puede asignar, y 2^S es el tamaño más grande, que coincide con la memoria del sistema.

Inicialmente, la memoria del sistema se considera como un bloque libre único de tamaño 2^S. Para cada potencia de 2 entre 2^I y 2^S se mantiene una lista de bloques libres de dicho tamaño, que inicialmente están vacías, excepto la lista de los bloques de tamaño 2^S. La lista de bloques libres de tamaño 2^i recibe el nombre de **lista** i.

Cuando se produce una petición de bloque de tamaño b, se busca un bloque de tamaño 2^i, donde i es el entero más pequeño tal que $2^{i-1} \leq b \leq 2^i$. Si no existe un bloque de tamaño 2^i con esas características, se escoge un bloque de tamaño 2^{i+1} para dividirlo en dos bloques iguales de tamaño 2^i, uno de los cuales se asigna y el otro permanece libre situándose en la lista i. En el caso de no existir bloques de tamaño 2^{i+1} libres, se divide un bloque de tamaño 2^{i+2} en dos bloques de 2^{i+1}, uno queda libre, y el otro se divide en dos bloques de tamaño 2^i. En caso de no existir bloques libres de tamaño 2^{i+2}, el proceso continúa hasta encontrar un bloque disponible. Si no existe bloque alguno, no se puede asignar la memoria.

En este esquema, si un bloque de tamaño 2^{i+1} empieza en la posición p, los dos bloques resultantes de la división de tamaño 2^i empiezan en las posiciones p y $p+2^i$, respectivamente. Se dice que ambos bloques son **compañeros**. Cuando se libere un bloque de tamaño 2^i y esté libre su compañero, ambos se combinan en un bloque de tamaño 2^{i+1}, para satisfacer solicitudes más grandes.

La figura 1.16 muestra un ejemplo del funcionamiento de este esquema en un sistema con una memoria de 1 MiB. Inicialmente se produce una petición para un proceso $P1$ de 45 KiB. El bloque inicial de la memoria se divide en dos compañeros de 512 KiB, el primero se divide a su vez en dos bloques de 256 KiB, que se subdividen en dos bloques de 128 KiB, para finalmente dividirlos en dos de 64 KiB, uno de los cuáles se asigna al proceso. A continuación se produce una petición de 110 KiB, que se le asigna el bloque de 128 KiB que se creó anteriormente. Este proceso de división de bloques continúa mientras sea necesario.

Cuando empiezan a producirse liberaciones de memoria, se van fusionando los huecos. Así, cuando el $P5$ finaliza libera un bloque de 128 KiB, que tiene a su compañero libre, fusionándose ambos en un bloque de 256 KiB. Éste presenta también a su compañero libre, por lo que pueden fusionarse en un único bloque de 512 KiB.

Este sistema presenta un esquema de asignación de memoria dinámico con un grado de multiprogramación flexible, al igual que el esquema de particiones variables. Con el fin de simplificar la gestión del espacio libre y asignado, la memoria se asigna en bloques de ciertos tamaños, asemejándose al esquema de particiones fijas, presentando también fragmentación interna.

1.9. Resumen

El administrador de la memoria es la parte del sistema operativo que se encarga del manejo de ésta. Entre las labores que debe realizar están: controlar las zonas de memoria ocupadas y libres, asignar memoria a los procesos, retirársela, controlar en que zona reside cada uno, etc.

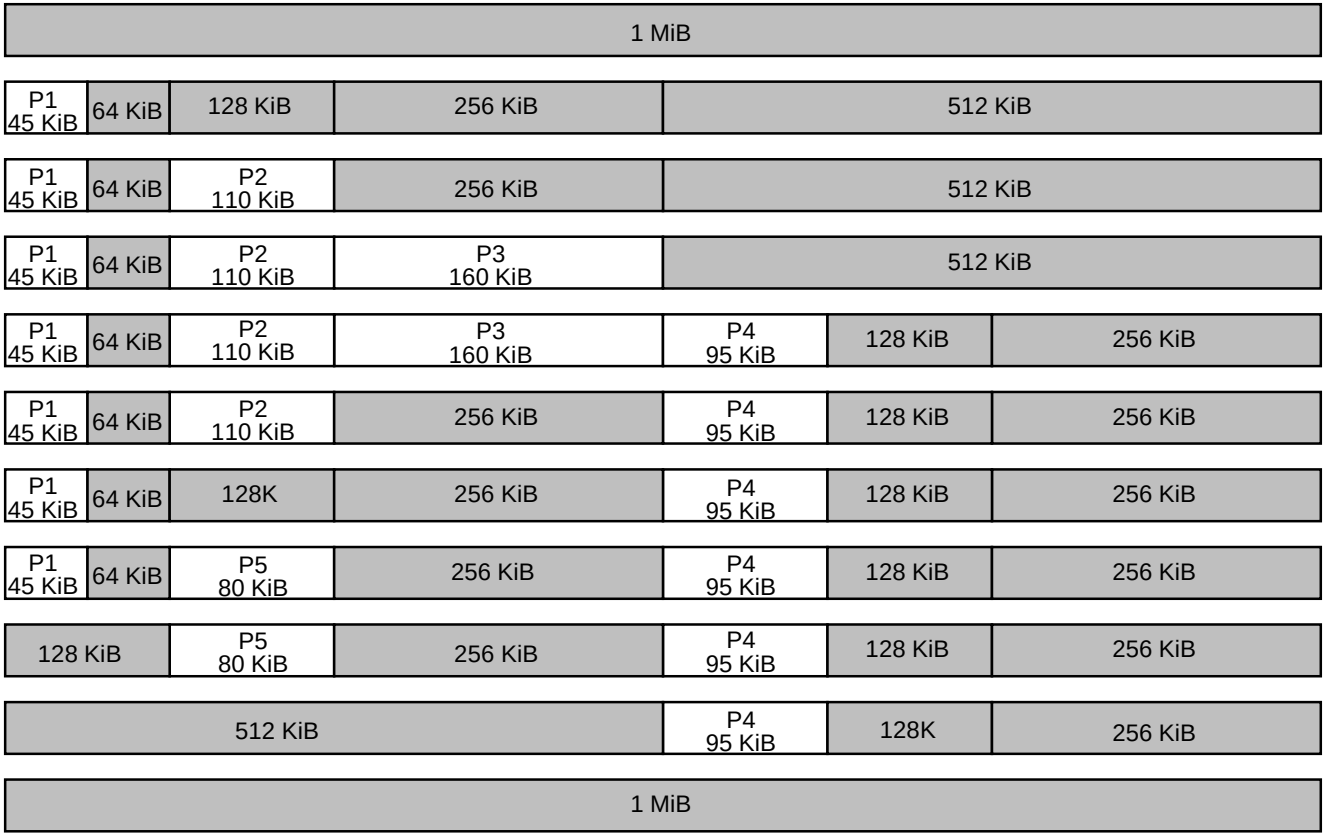

Figura 1.16: Sistema compañero

Los sistemas pueden adoptar distintos esquemas de asignación de la memoria principal, éstos se pueden clasificar en dos grandes grupos, los de asignación contigua, que exigen que la imagen del proceso se cargue en una zona de memoria contigua; y los de asignación no contigua, que no presentan la exigencia anterior. Dentro del primer grupo se encuentran los sistemas de monoprogramación y los de multiprogramación con particiones. La paginación y la segmentación son esquemas de asignación no contigua que se usan en los sistemas de memoria virtual que se estudiarán en Sistemas Operativos II.

En los sistemas de monoprogramación sólo residen en memoria principal el sistema operativo y un proceso de usuario, sin embargo los sistemas de multiprogramación son más complejos porque varios procesos de usuario comparten la memoria, en este caso existen diversas formas de asignarla. Se pueden crear una serie de particiones de tamaño fijo en las que se carguen los procesos, o bien se pueden crear las particiones a medida que se necesitan y adaptándose al tamaño de los procesos.