CENTRO UNIVERSITARIO
DE TECNOLOGÍA
Y ARTE DIGITAL

Sistemas Operativos

4. Gestión de memoria

Javier García Algarra
javier.algarra@u-tad.com
Miguel Ángel Mesas
miguel.mesas@u-tad.com
Carlos M. Vallez
carlos.vallez@u-tad.com

2022-2023

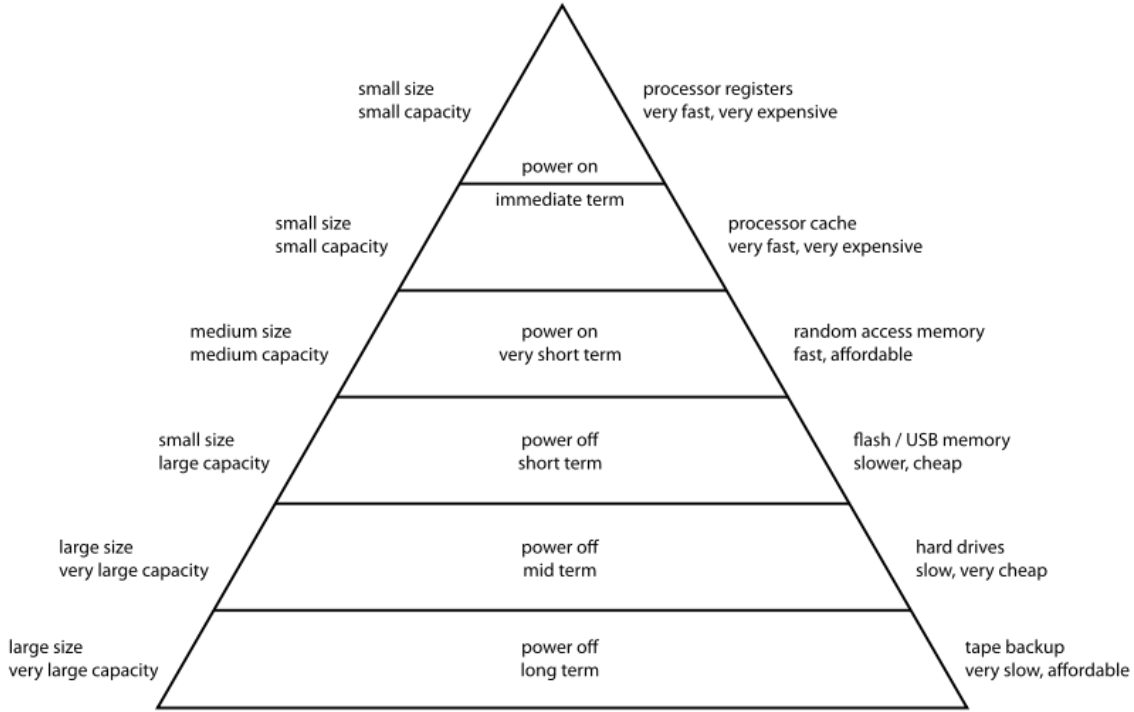
Introducción

La memoria es el elemento que almacena los datos y el programa en el modelo de von Neumann. Aparentemente no parece que el Sistema Operativo tenga mucho que hacer con un elemento que es hardware.

Sin embargo, en un sistema multiproceso, el SO tiene que encargarse de que todos los procesos tengan la memoria necesaria y llevar la contabilidad de dónde están cargados. Además debe impedir que un proceso acceda a la memoria de otro o que por error corrompa su propio ejecutable.

Añádase a esto que la velocidad de acceso a la memoria RAM no ha aumentado en la misma proporción que la velocidad de proceso de las CPU y que en un PC de sobremesa podemos tener en cualquier momento un centenar de procesos que no cabrían en la memoria física. El SO resuelve todos estos problemas gestionando una jerarquía de memoria muy sofisticada.

Jerarquía de memoria



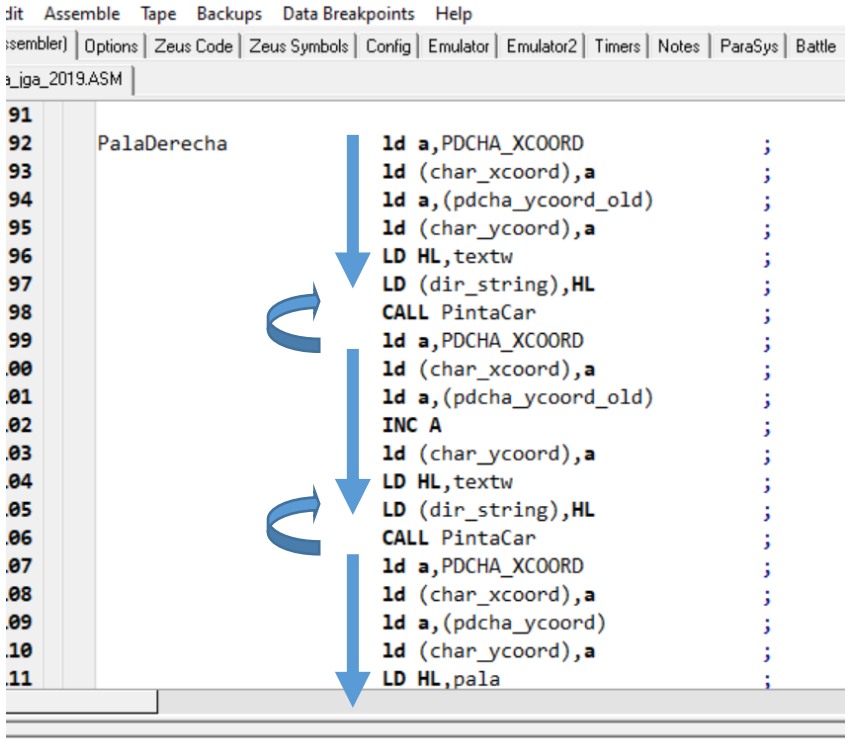
Jerarquía de memoria

El funcionamiento de la jerarquía de memoria se basa en dos posibilidades estadísticas del código: **proximidad o localidad espacial** y **proximidad o localidad temporal**.

- La **proximidad espacial** se produce porque el programa se ejecuta de forma lineal la mayor parte del tiempo y por tanto las instrucciones de código máquina son consecutivas y los datos también suelen residir en zonas acotadas (vectores, tablas)

- La **proximidad temporal** se produce por la repetición de secuencias de código en bucles y llamadas a funciones.

Se denomina **conjunto de trabajo** al fragmento de código y datos que se acceden de forma repetida durante un periodo breve.



Flujo de ejecución

Jerarquía de memoria

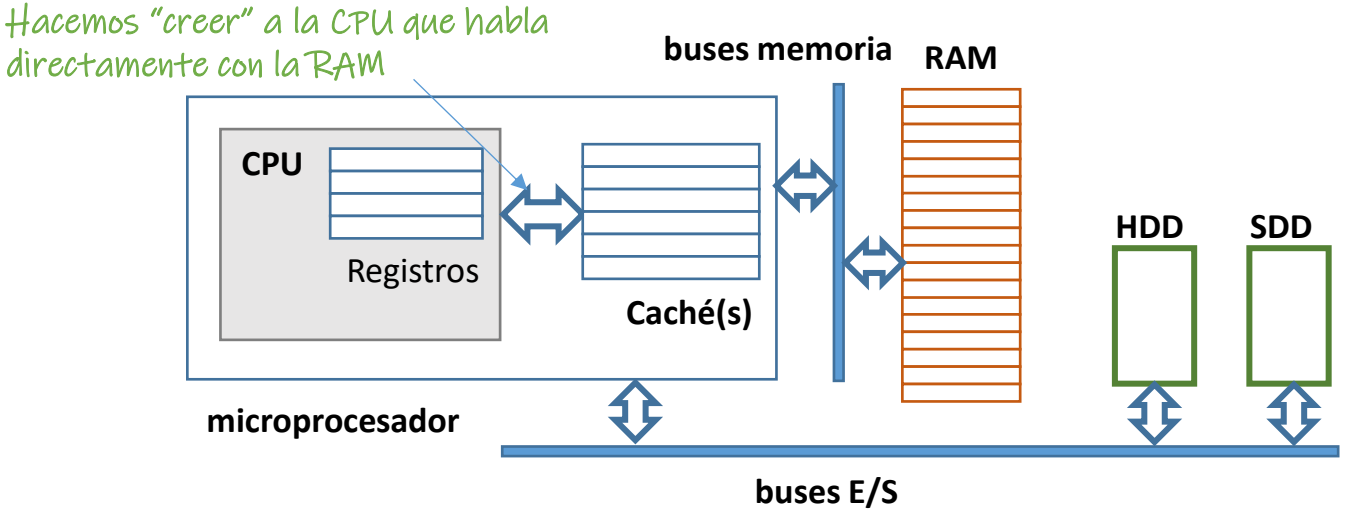


Tabla 1.1 *Valores típicos de la jerarquía de memoria.*

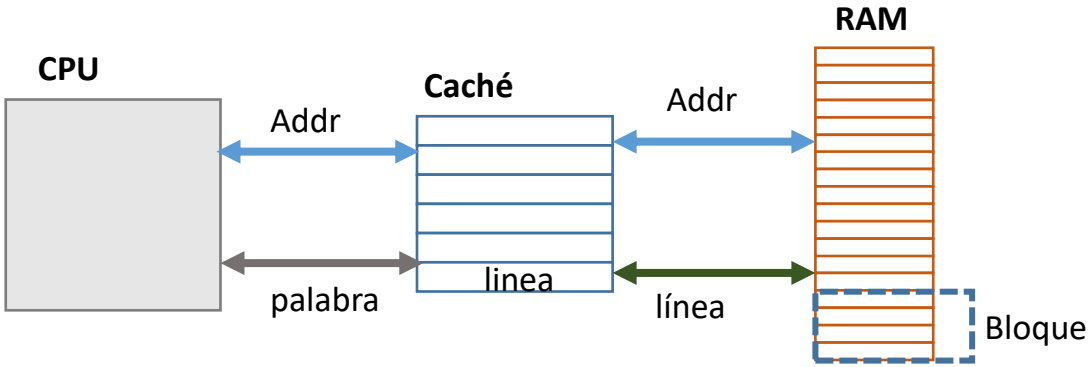
Nivel de memoria	Capacidad	Tiempo de acceso	Tipo de acceso
Registros	64 a 1024 bytes	0,25 a 0,5 ns	Palabra
cache de memoria principal	8 KiB a 8 MiB	0,5 a 20 ns	Palabra
Memoria principal	128 MiB a 64 GiB	60 a 200 ns	Palabra
Disco electrónico SSD	128 GiB a 1 TiB	50 µs (lectura)	Sector
Disco magnéticos	256 GiB a 4 TiB	5 a 30 ms	Sector

De Miguel, Pérez Costoya: "Sistemas Operativos", UPM, 2016

Memoria caché

La memoria caché es una memoria muy pequeña y muy rápida que reside en el mismo sustrato de silicio que la CPU, aunque no forma parte de ella. En la arquitectura de von Neumann la CPU se relaciona directamente con la memoria usando los buses de direcciones, datos y control. Cuando la caché está presente, "engaña" a la CPU, capturando las señales de esos buses y dialogando con la CPU como si fuese la memoria convencional.

Por otra parte se conecta a la RAM. Mientras que la CPU lee y escribe "palabras" de la RAM, la memoria caché intercambia "bloques" con la RAM, conjuntos de 8, 16, 32… bytes, cuyo tamaño coincide con el de la "línea" o unidad en la que se organiza internamente la caché.



Memoria caché

El acceso a la RAM es muy lento en comparación con la velocidad del procesador. La idea es colocar una memoria muy rápida oculta (*caché* en francés) entre la CPU y la RAM. Si la mayor parte del tiempo los datos o el código están en esa memoria, aumentaremos sensiblemente la capacidad de proceso. Cuando el dato está en la caché se registra un acierto ("hit") mientras que si no lo está se produce un fallo ("miss").

Supongamos que tenemos una caché con un tiempo de acceso de 2ns y una RAM con tiempo de acceso de 50ns. Si la tasa de acierto es de un 90%, ¿cuál es el tiempo efectivo de acceso a memoria?

Es un problema clásico de probabilidad, el tiempo efectivo es el valor medio:

$$t_{ef} = t_{hit} \times P(hit) + t_{miss} \times P(miss) = t_{hit} \times P(hit) + t_{miss} \times (1-P(hit))$$

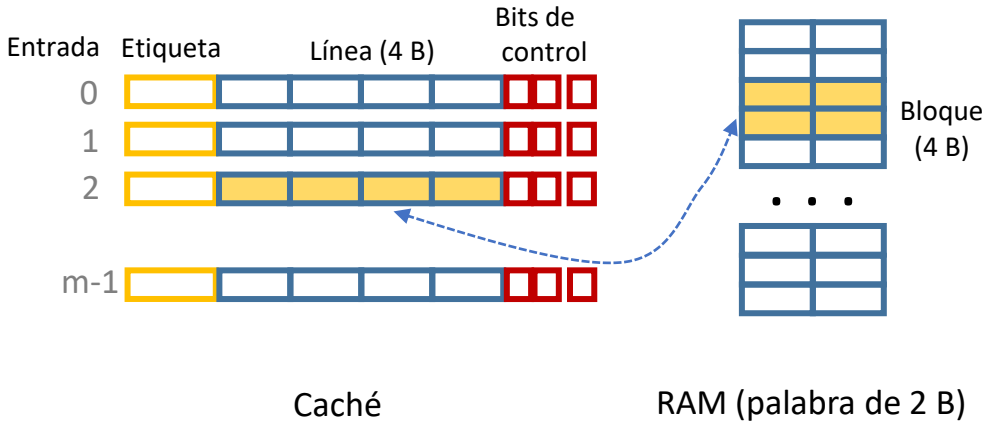
El tiempo de tratamiento de fallo se iguala al de acceso a la memoria RAM suponiendo que la circuitería de la caché es muy rápida

$$t_{ef} = 2 \times 0.9 + 50 \times 0.1 = 6.8 \text{ ns}$$

Memoria caché

La realidad es más compleja porque la transferencia entre caché y RAM es en bloques. Supongamos que los bloques son de 4 palabras, en ese caso el tiempo de tratamiento de fallo subiría a 200 ns y el tiempo eficaz de acceso a la memoria pasaría a ser de 21,8 ns. Aunque vemos un significativo aumento de tiempo, el acceso a caché sigue siendo inferior al tiempo de acceso a RAM.

La caché se organiza en “líneas” o conjuntos de palabras que se transfieren desde o hacia la RAM de una vez. El tamaño de la “línea” es igual al tamaño en bytes de lo que llama “bloque” en memoria. Esto se cumple siempre. Adicionalmente cada entrada de la caché tiene un campo llamado “etiqueta” y una serie de bits de control.



Memoria caché

EJEMPLO

Una CPU tiene bus de direcciones de 16 bits, una memoria caché de 32 entradas y un tamaño de línea de 4 bytes. ¿Cuál es el tamaño del bloque? ¿Cuál es la memoria RAM máxima posible? ¿cuántos bloques alberga la memoria RAM suponiendo que se ha instalado el máximo posible?.

La primera pregunta es trivial porque el enunciado indica que el tamaño de línea es de 4 bytes, por tanto el tamaño de bloque es también de 4 bytes.

La memoria física máxima posible la determina el número de bits del bus de direcciones. Con 16 bits podemos direccionar 2^{16} bytes (recuerda que la unidad mínima de lectura/escritura en la RAM es el byte).

$2^{16} = 2^6 \times 2^{10} = 64$ kB (la memoria máxima típica del Spectrum)

Como cada bloque tiene 4 bytes, el número de bloques de esa memoria es $2^{16}/2^2 = 2^{14}$ bloques (exactamente 16384 bloques pero puedes dejarlo en potencia de 2 en el examen)

El dato del número de entradas de la caché es irrelevante

Memoria caché

Como el número de entradas de la caché es mucho menor que el de bloques de la memoria, es necesario establecer un mecanismo de correspondencia que permita saber qué bloques están cargados y en qué entrada. Hay tres modos: correspondencia directa, correspondencia asociativa y correspondencia asociativa por conjuntos.

Método de correspondencia directa

Es el más simple, cada bloque solo puede instanciarse en una entrada determinada de la caché. Permite construir cachés con circuitería muy sencilla. Supongamos que la caché tiene k entradas/líneas, siempre potencia de dos y que la memoria tiene m bloques (m >> k).

El bloque j de memoria RAM se instancia en la entrada `j mod k`.

Tenemos una RAM de 32 kB y una caché de 32 líneas. El tamaño de la línea son 4 B. ¿En qué entrada se instancia el bloque 2004?

Solución = 2004 mod 32 = 20.

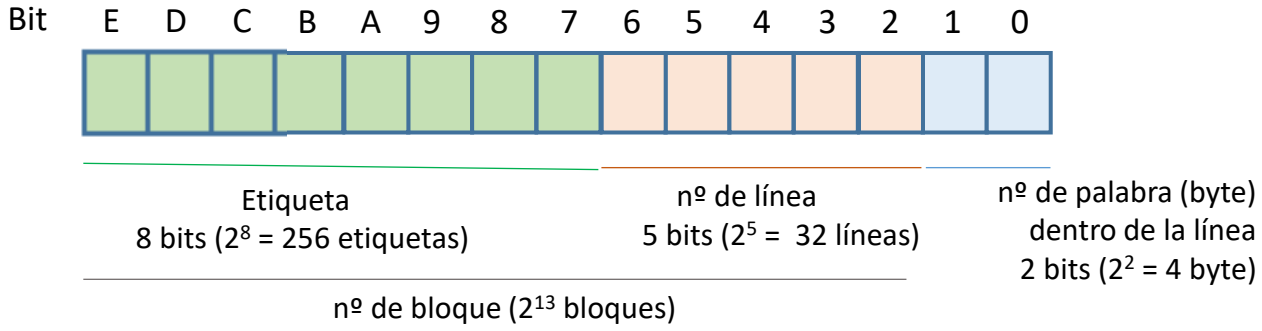
Es la misma línea en la que se cargarían el bloque 20 o el 84. ¿Cómo sabe la caché qué bloque es el que tiene cargado?.

Memoria caché

Método de correspondencia directa (continuación)

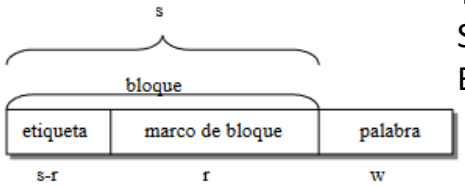
El campo Etiqueta (“label”) es el que permite saber a la caché qué bloque tiene cargado. Para la correspondencia directa, el campo etiqueta del bloque j se obtiene como el cociente de esa cifra entre el número de entradas. Para nuestro ejemplo: Etiqueta = 2004 div 32 = 62
Para una misma línea, el número de etiquetas es el número de bloques total entre el número de entradas de la caché.

Para este método de correspondencia existe una relación sencilla entre la dirección de memoria, el número de entrada en la caché y la etiqueta.



Memoria caché

Método de correspondencia directa (continuación)



2^w palabras/bloque
2^s bloques de Mp
2^r marcos de bloque en Mc ($2^r = m$)
2^{s-r} veces contiene Mp a Mc

Tamaño del bloque 4 Bytes= 2^2 => w=2

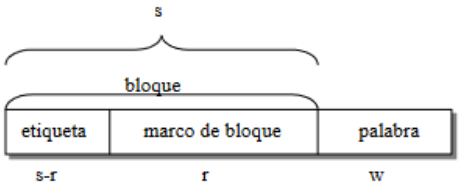
Tamaño Mp= 32 KB = 2^{15} Bytes
Si cada bloque es de 4 Bytes tengo 2^{15} / 2^2 = 2^{13} Bloques en la RAM. s=13

Tamaño Caché = 32 x 4= 2^7 Bytes
Numero de marcos (líneas)= Tamaño cache/ Tamaño del bloque
numero de marcos (líneas) =2^7/ 2^2 = 2^5
Luego r=5

Etiqueta= s-r= 13 -5 = 8

Memoria caché

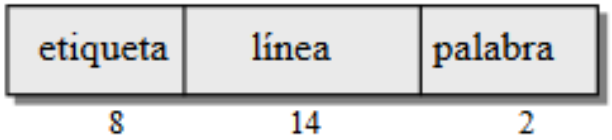
Método de correspondencia directa (continuación) Otro ejemplo



2^w palabras/bloque
2^s bloques de Mp
2^r marcos de bloque en Mc ($2^r = m$)
2^{s-r} veces contiene Mp a Mc

Tamaño del bloque 4 Bytes= 2^2 => w=2

Tamaño Caché = 64KB= 2^{16} Bytes
Numero de marcos (líneas)= Tamaño cache/ Tamaño del bloque
numero de marcos (líneas) =$2^{16}/ 2^2 = 2^{14}$
Luego r=14



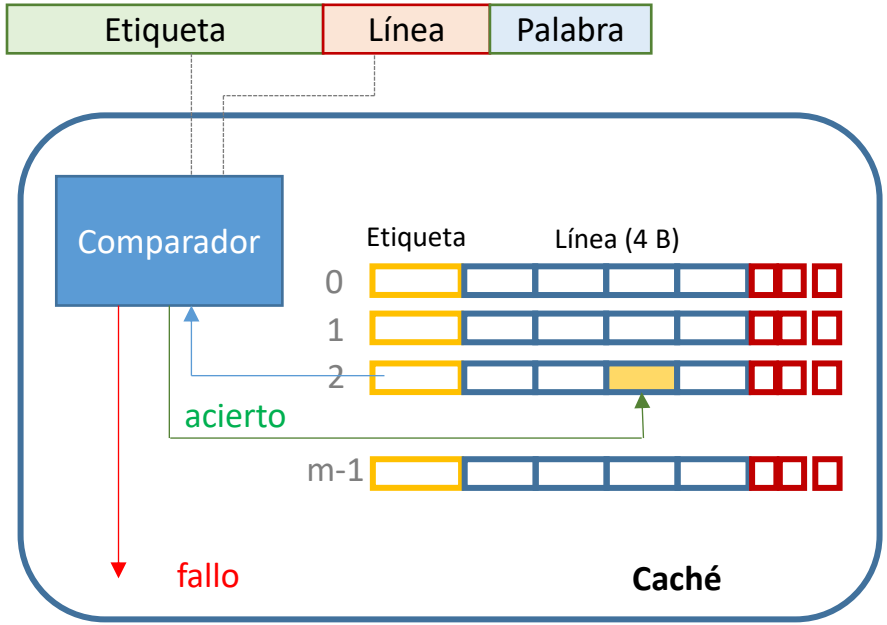
Tamaño Mp= 16MB= 2^{24} Bytes
Si cada bloque es de 4 Bytes tengo $2^{24}/ 2^2 = 2^{22}$ Bloques en la RAM. s=22

Etiqueta= s-r= 22-14 = 8

Memoria caché

Método de correspondencia directa (continuación)

Dirección de memoria



La circuitería es muy simple. Un circuito compara el contenido del campo Etiqueta de la línea indicada en la dirección de memoria con el propio contenido de Etiqueta en dicha dirección. Si son diferentes, el bloque no está cargado, se genera un fallo y otro circuito (que no incluimos) se encarga de actualizar la línea. Si hay acierto se selecciona la palabra que indican los bits de menos peso de la dirección.

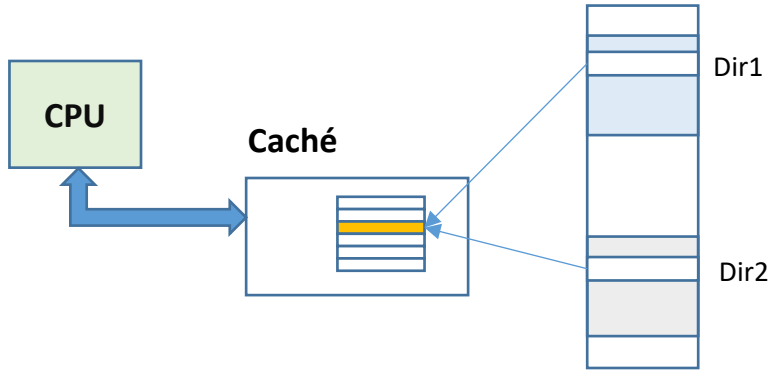
Memoria caché

Método de correspondencia directa (continuación)

Inconvenientes del método:

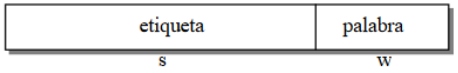
- Puede haber líneas que no se usan mientras otras se están actualizando continuamente.
- *Thrashing*. Si tenemos mala suerte, un programa puede entrar en un bucle que accede a dos bloques que se instancian en la misma línea. La caché produciría fallos de forma continua degradando en gran medida el tiempo de acceso.

```
Bucle   LD A, (Dir1)
        INC
        LD (Dir2),A
        JP Bucle
```



Los dos bloques se cargan en la misma línea

Memoria caché



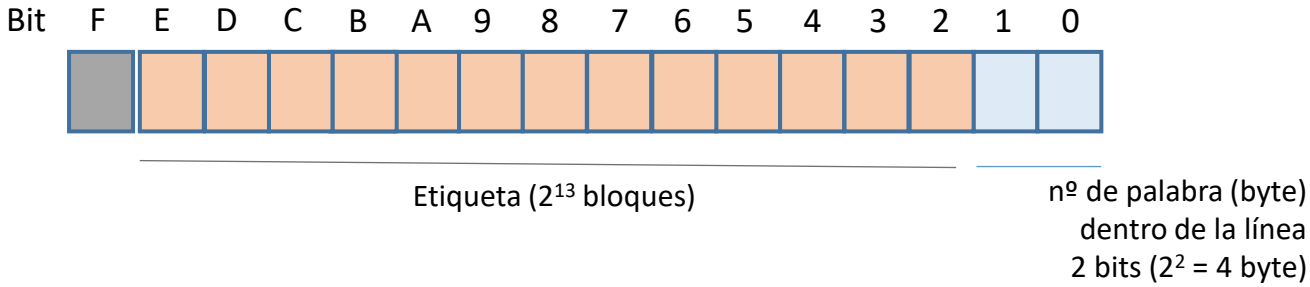
2^s bloques de Mp

2^r marcos de bloque de Mc

Método de correspondencia asociativa

En la correspondencia asociativa cualquier bloque puede instanciarse en cualquier línea de la caché, con lo que se consigue una ganancia estadística en su uso frente a la correspondencia directa. La etiqueta tiene que permitir numerar todos los bloques.

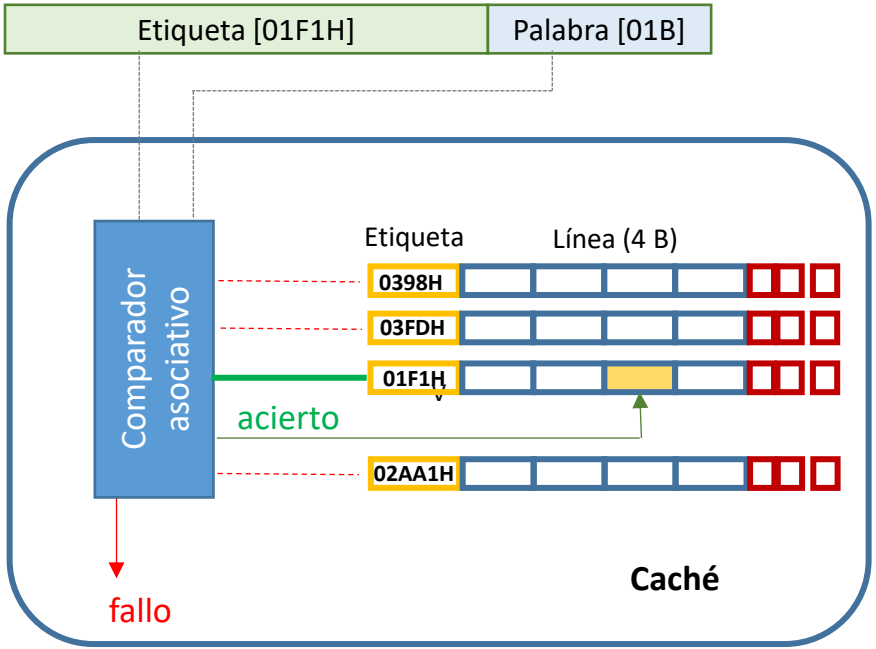
Para el mismo ejemplo de máquina, 32 KB de RAM, con 16 bits de direcciones, caché de 32 líneas y 4 bytes por línea, las etiquetas tienen que tener 13 bits. En este método no se sabe el número de línea de antemano porque puede ser cualquiera.



Memoria caché

Método de correspondencia asociativa (continuación)

Dirección de memoria



La circuitería es mucho más compleja. La caché se comporta como una memoria asociativa. Hay un circuito que permite comparar los datos del campo Etiqueta de la dirección con todas las etiquetas instanciadas de forma simultánea y devuelve la línea que contiene el dato o fallo si el bloque no está cargado

Memoria caché

Método de correspondencia asociativa (continuación)

Inconvenientes del método:

- El principal es la complejidad electrónica. El comparador tiene que ser capaz de comparar los m bits de la etiqueta, para cada una de las n líneas de la caché. Esto se hace mediante circuitos combinacionales que ocupan superficie muy valiosa de la oblea del microprocesador y no escala para los tamaños actuales de memoria.

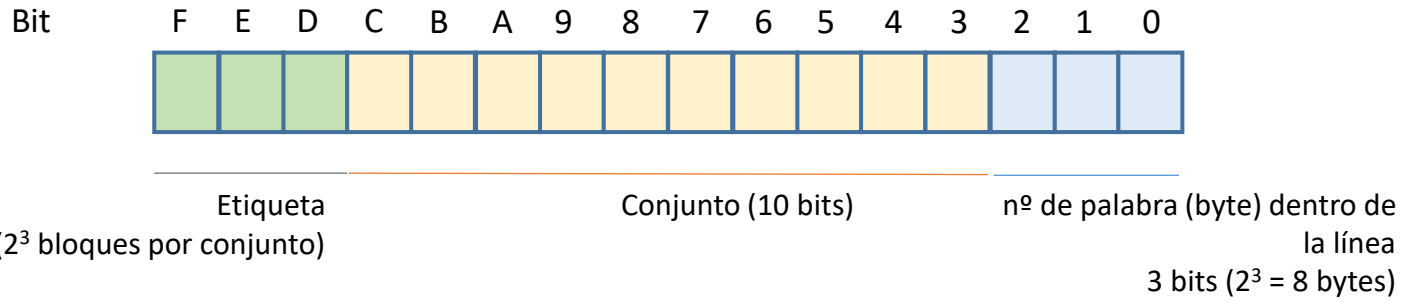
Método de correspondencia asociativa por conjuntos

El tercer y último método combina ventajas de la correspondencia directa y de la asociativa alcanzando un rendimiento muy parecido al de esta última. Los bloques se agrupan en conjuntos de tamaño fijo (potencia de dos) y cada conjunto se instancia siempre en las mismas líneas de caché. La flexibilidad surge porque dentro de las líneas del conjunto un bloque puede instanciarse en cualquiera de ellas. Hay solo un comparador asociativo por conjunto muy simple puesto que solo tendrá que realizar las comparaciones de las líneas del conjunto seleccionado.

Memoria caché

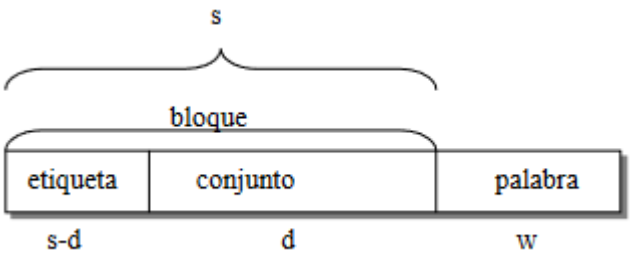
Método de correspondencia asociativa por conjuntos (continuación)

Las direcciones tienen tres campos. Los bits de menor peso como siempre son la palabra dentro de la línea [es el único campo común a los tres métodos]. Los bits intermedios permiten seleccionar el conjunto de bloques. Si C es el conjunto que viene en la dirección, y nc el número de conjuntos de la caché, se mapea en C mod nc. Cada conjunto de la caché agrupa un número de líneas igual al número de bloques de un conjunto. Los de mayor peso son la etiqueta, que nos permite saber de qué bloque dentro de un conjunto se trata.
Para el mismo ejemplo de máquina con 16 bits de direcciones, supongamos conjuntos de 8 bloques (3 bits de dirección) y líneas de 8 bytes (3 bits). El número de conjuntos es 2^{10} (1024)



Memoria caché

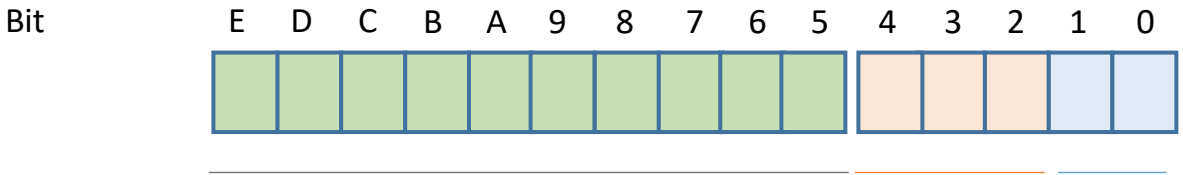
Método de correspondencia asociativa por conjuntos (continuación)



Tamaño del bloque 4 Bytes= 2^2 => w=2

Bus 16 bits => Tamaño Mp= 32 KB = 2^{15} Bytes
Si cada bloque es de 4 Bytes tengo 2^{15} / 2^2 = 2^{13} Bloques en la RAM. => s=13

32 líneas que dividiré en 8 grupos
V= 2^d => 8= 2^d => d=3 Con 3 bits represento los 8 grupos



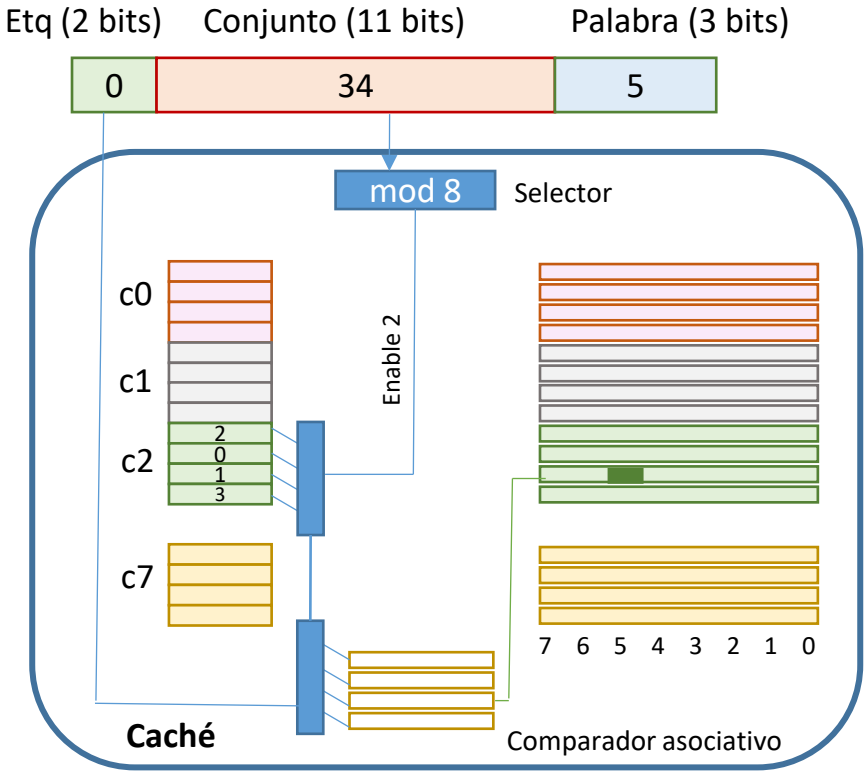
Etiqueta 10 bits
2^{15} Bloques
Pero como leo 2 palabras 2^{13}
3 bits los codifico con el grupo, luego necesito 10 bits más en la etiqueta

Conjunto (3 bits)

nº de palabra (byte) dentro de la línea
2 bits (2^2 = 4 bytes)

Memoria caché

Método de correspondencia asociativa por conjuntos (continuación)



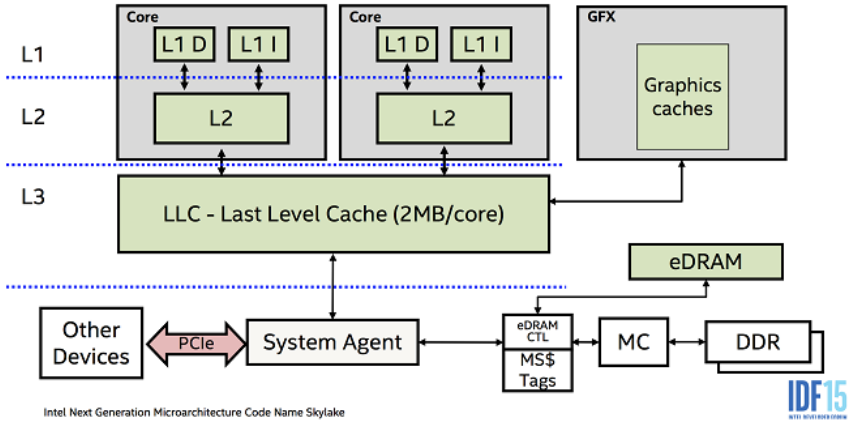
En este ejemplo de máquina de 16 bits, la caché tiene 32 líneas de 8 bytes [3 bits] cada una y los conjuntos son de 4 bloques [la etiqueta tiene 2 bits].

El número de conjunto tiene por tanto 11 bits. Con este número un primer selector escoge el conjunto de líneas de la caché haciendo módulo con el número de conjuntos en caché. Estas etiquetas se presentan a la entrada del pequeño comparador asociativo que localiza la línea en que se ha instanciado y dentro de ella, extrae el dato indicado por el campo palabra.

Memoria caché

Caché de varios niveles

En los microprocesadores modernos no hay una sola caché, sino varios niveles (3 en los micros Intel). El nivel 1 es el más pequeño y más rápido. Cada nivel actúa como caché del nivel previo. La comunicación entre las cachés es mucho más rápida que con la RAM porque están en el mismo circuito que la CPU.
La gestión se complica cuando hay varios núcleos por microprocesador.



Esquema de cachés en Intel. Nótese que hay dos cachés L1 (para código y datos) y que la de nivel 3 (LLC) es compartida.

Memoria caché

Políticas de reemplazo

Cuando se produce un fallo la caché debe decidir qué línea expulsar para reemplazarla por la que se ha solicitado. Esto lo realiza la circuitería de la caché, que usa para ello los bits de control ya mencionados.

Si la caché es de ***correspondencia directa***, la política es trivial, porque cada bloque solo puede cargarse en una línea concreta. Si el bit de esa línea en la caché indica que está "sucia" la caché debe escribir el contenido de la línea expulsada en su bloque correspondiente de la RAM antes de reemplazarlo. Si la caché es ***asociativa***, el bloque puede instanciarse en cualquier línea. Si es ***asociativa por conjuntos***, la decisión afectará solo a las líneas del conjunto correspondiente:

- Reemplazo aleatorio: Se decide una línea al azar. Simplifica la circuitería.
- FIFO: Se reemplaza la línea que lleve más tiempo cargada. Hace falta un timestamp por línea.
- LRU (Least Recently Used): Se expulsa la línea que lleva más tiempo sin ser referenciada.
- LFU (Least Frequently Used): Se expulsa la que se haya referenciado menos veces.

Memoria caché

Políticas de escritura

Cada vez que un dato de la caché se escribe, queda desactualizada la RAM o de los otros niveles de caché. Hay que proceder a su sincronización. Puede hacerse de distintas maneras.

- **Write Through**: Cada vez que hay una escritura en caché, se reescribe el bloque en la RAM. Es muy simple pero se pierde la ventaja de tener una caché en operaciones de escritura.

- **Write Back**: La línea solo se escribe en RAM si está sucia en el momento de ser reemplazada. Genera mucho menos trasiego y como precio que hay que pagar, la circuitería es más compleja porque hay que mantener y comprobar el bit “dirty”.

En cualquiera de los casos, la escritura se hace siempre por líneas completas.

Para saber más https://youtube.com/playlist?list=PLbtzT1TYeoMgJ4NcWFuXpnF24fsiaOdGq

Memoria real

La memoria real o memoria física es la RAM instalada en el ordenador. Aunque los Sistemas Operativos de propósito general trabajan con memoria virtual, que se trata a continuación, hay situaciones en las que puede resultar necesario trabajar con procesos que estén cargados completamente en memoria. Los sistemas de soporte vital o *misision critical* entran dentro de esta categoría.

El problema que tiene que resolver el Sistema Operativo es cargar los distintos procesos en cada momento y hacerlo en zonas contiguas de memoria (o el menos contiguos en los segmentos).

En los sistemas monoproceso era muy simple, pero la situación se vuelve mucho más complicada cuando hay decenas o centenas de procesos que tienen que cargarse en memoria simultáneamente. Una solución muy primitiva fue el *swapping,* el Sistema Operativo concedía toda la memoria física al proceso en ejecución y copia a disco los datos del proceso expulsado.

Memoria real

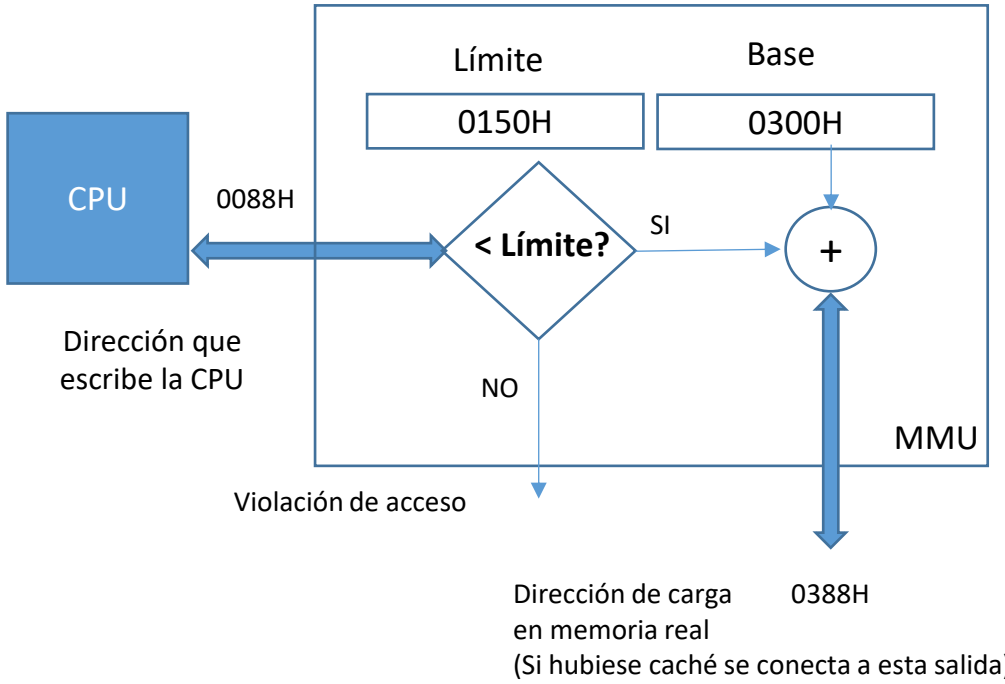
En estos sistemas que trabajan en modo real, tiene que haber un mecanismo HW de protección que evite que un proceso acceda a la dirección de otro proceso, porque esa comprobación hay que hacerla en toda operación de lectura o escritura. Esta función la realiza la MMU (Memory Management Unit) que se incluye en el microprocesador. Un método sencillo de conseguirlo consiste en tener un registro base con la dirección inicial del proceso y otro límite, en cada acceso a RAM la MMU comprobará por hardware que la dirección no sale de ese rango, de lo contrario produce una interrupción que atenderá el Sistema Operativo (por lo general matará al proceso que intenta el acceso ilegal).

El kernel mantiene una tabla con todos estos pares base/límite y actualiza los registros de la MMU en cada cambio de contexto. Esta tabla permite reubicar los procesos cambiando la base. Las direcciones que escribe la CPU se recalculan por la MMU en cada acceso.

Memoria real

base A	**0300H**
límite A	**0150H**
base B	**01F0H**
límite B	**02E0H**

Tabla de asignación de memoria del SO



Dirección que escribe la CPU

El proceso A está cargado a partir de la dirección 0300H y en ejecución. La MMU tiene cargados los datos del proceso A, comprueba en cada acceso que no exceden los límites y traduce a dirección absoluta de carga. Cambiando la Base, se reubica el proceso en otra zona de forma transparente al programador.

Memoria real

Técnicas de asignación de memoria

En los sistemas con memoria real, hay que proceder al reparto entre todos los procesos cargados cuyos tamaños pueden ser muy diferentes. Cuando hay que cargar un nuevo proceso, es muy improbable que haya un hueco del tamaño exacto. Eso producirá fragmentación, se generan pequeños huecos no ocupados pero que no pueden utilizarse con lo que el uso de la memoria no es óptimo.

Además, el sistema tiene que saber qué zonas de memoria se encuentran libres y eso implica que tiene que gastar una parte de dicha memoria en guardar ese registro. Hay dos técnicas, bitmap y listas encadenadas.

El mapa de bits implica dividir la memoria en pequeñas zonas (por ejemplo 32 bits) y marcar en un vector de bits si están ocupadas o no. Necesitamos 1 bit por cada 32 bits, con lo que en un sistema de 4GB el bitmap ocuparía 128 MB.

Memoria real

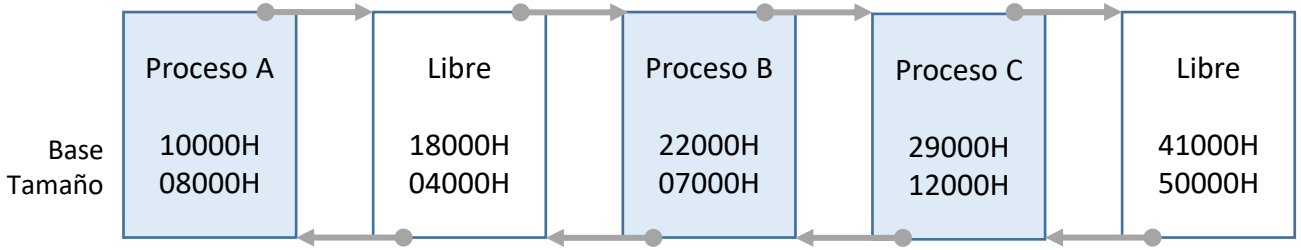
Técnicas de asignación de memoria

El bitmap tiene un problema de rendimiento. Cuando el Sistema Operativo tiene que cargar un nuevo proceso del tamaño que sea, tiene que recorrer el bitmap desde el principio hasta entrar el hueco.

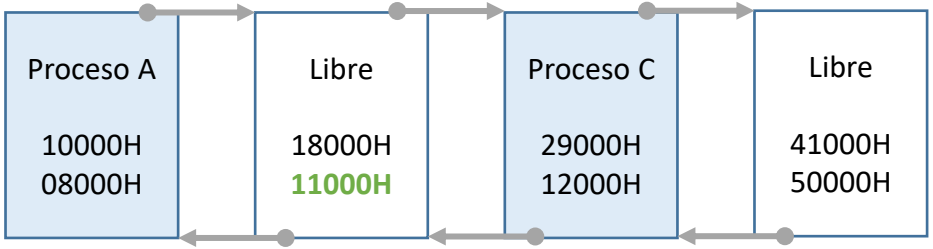
Una solución mejor es utilizar una lista doblemente encadenada que guarda los datos de los fragmentos de memoria ocupados por procesos y los que están libres.

Es mucho más rápido recorrer una lista de este tipo que el bitmap y además consume mucho menos espacio en memoria. El inconveniente es que es memoria dinámica que hay que gestionar desde e Sistema Operativo y puede ser complejo para equipos empotrados de bajas prestaciones, que usan pocos procesos y emplean el mapa de bits.

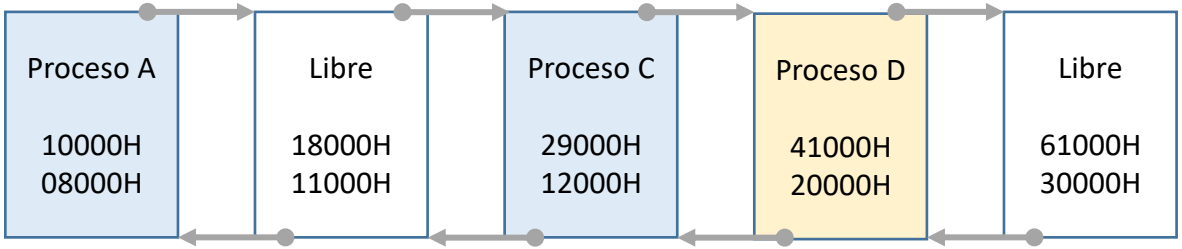
Memoria real



Situación inicial



Termina B (su espacio se asigna al bloque libre contiguo)



Se carga D, que ocupa 20000H B de memoria

Memoria real

Algoritmos de asignación de memoria real

En el ejemplo anterior se ha asignado el primer hueco en el que cabía el proceso. Es lo que se denomina **First Fit** pero hay otros criterios.

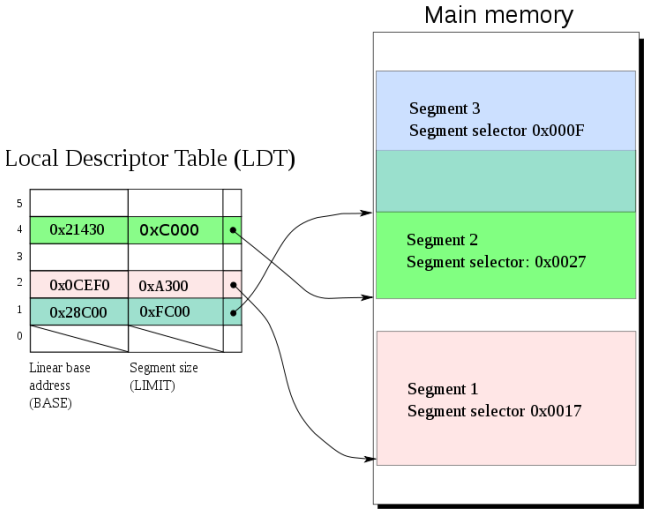
- **Best Fit**. Se busca el hueco más pequeño en el que cabe el proceso. Obliga a recorrer toda la lista, pero reduce al máximo el espacio desaprovechado.
- **Next Fit**. Es igual que el Fist Fit, pero en lugar de recorrer la lista desde el inicio se empieza desde el último espacio asignado, para conseguir un reparto más equitativo y reducir de ese modo la fragmentación.
- **Worst Fit**. En lugar de buscar el hueco más pequeño disponible, se asigna el más grande. De este modo se evita generar muchas pequeñas zonas desaprovechadas. Tiene el mismo inconveniente que Best Fit, hay que recorrer toda la lista.
- **Quick Fit.** En lugar de una lista se mantienen varias con huecos estandarizados de (2 kB, 10 kB, 40 kB, …). Al terminar un proceso hay que hacer un merge con el espacio de memoria libre más próximo.

El sistema operativo usa estos mismos criterios para gestionar el heap de un único proceso.

Segmentación

La segmentación es una técnica de reubicación que emplea la idea de puntero a una base reubicable, y permite usar distintos segmentos de la memoria para el código, los datos o el stack. En el caso de la arquitectura x86, hay 4 segmentos, además de los tres definidos hay uno adicional de uso libre.

La segmentación tiene la ventaja de añadir una capa adicional de protección, porque el programa no podrá escribir en la zona de código o ejecutar la zona de datos. Cada proceso tiene su propia tabla de segmentos (base y límite) en la zona de memoria del kernel. Aunque los micros Intel actuales mantienen los registros de paginación, los sistemas operativos como Linux o Windows no hacen uso de ella. Para evitarlo, simplemente marcan la dirección 0000H como base y el máximo del espacio de memoria virtual como límite en todos los segmentos.



Memoria virtual

Hoy en día solo trabajan con memoria real los sistemas empotrados muy pequeños (por ejemplo, Arduino) o aquellos de tiempo real que por requisitos de rendimiento necesitan todos los procesos cargados en cualquier momento, como en funciones de soporte vital.

Todos los demás funcionan con **memoria virtual**, concepto que no tiene nada que ver con la virtualización de equipos. Esta idea nació en los años 60 con una situación muy diferente a la nuestra. La memoria RAM era muy cara y los equipos multiproceso, simplemente no tenían capacidad para tener todos los procesos en ejecución completamente cargados.

Puesto que el *conjunto de trabajo* de un proceso es mucho más reducido que su imagen de memoria completa, se trata de cargar solo la parte estrictamente necesaria para la ejecución en un instante dado e ir actualizándola.

Memoria virtual

En los equipos con memoria virtual, la RAM actúa respecto al disco como la caché respecto a la RAM. Esta idea tuvo que solucionar obstáculos muy grandes para terminar siendo un éxito:

- El sistema operativo tiene que conseguir que esta gestión sea transparente al proceso y no condicione la programación. En consecuencia, se convierte en una de las funciones centrales del kernel.

- La traducción de direcciones virtuales a direcciones físicas (las que están realmente cargadas) tiene que ser muy rápida. Eso requiere que los microprocesadores dispongan de circuitos MMU (Memory Management Unit). El Sistema Operativo tendrá que manejar los detalles concretos de la MMU del modelo correspondiente.

- Los tiempos de acceso a disco magnético son unos 6 órdenes de magnitud superiores a los de la RAM, los fallos tienen que reducirse al mínimo inevitable

Memoria virtual

Todos los sistemas modernos usan la técnica de paginación. La memoria virtual, la memoria física y los accesos a y desde disco para cargar y escribir páginas se hacen usando páginas de tamaño fijo, un valor típico son 4kB, aunque puede variar por configuración y de sistema a sistema.

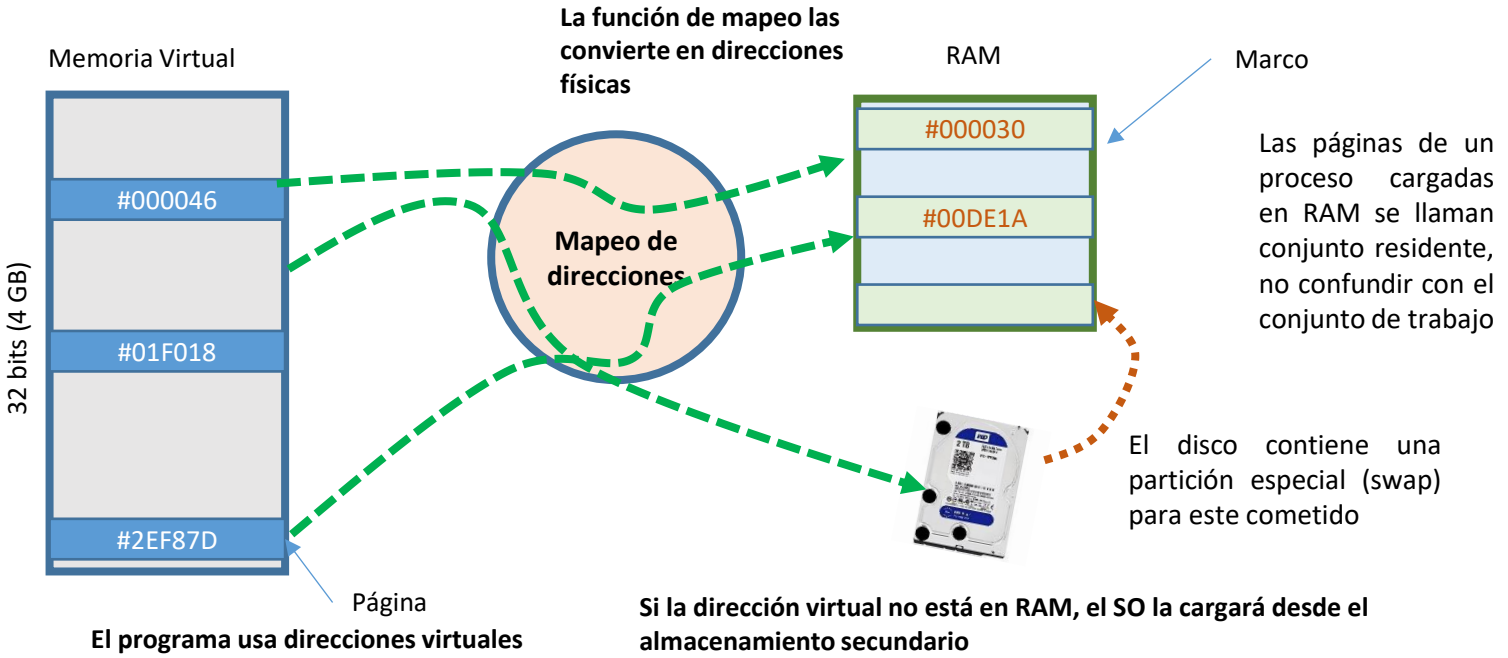
Los procesos ven un mapa de memoria virtual completo que se divide en “páginas”. Cada página virtual se instancia en un “marco” de la memoria física. El Sistema Operativo es el encargado de realizar la gestión y preparar los datos para el circuito que realiza las traducciones.

Cuando la página no está cargada en la RAM se dice que se ha producido un fallo de página y tiene que resolverse buscando un marco libre (o expulsando uno ocupado) en el que cargar dicha página desde disco. En el tiempo de tratamiento de un fallo de este tipo el micro puede ordenar del orden de 50 millones de instrucciones, por lo que el proceso se bloquea para dejar la CPU a otro.

Memoria virtual

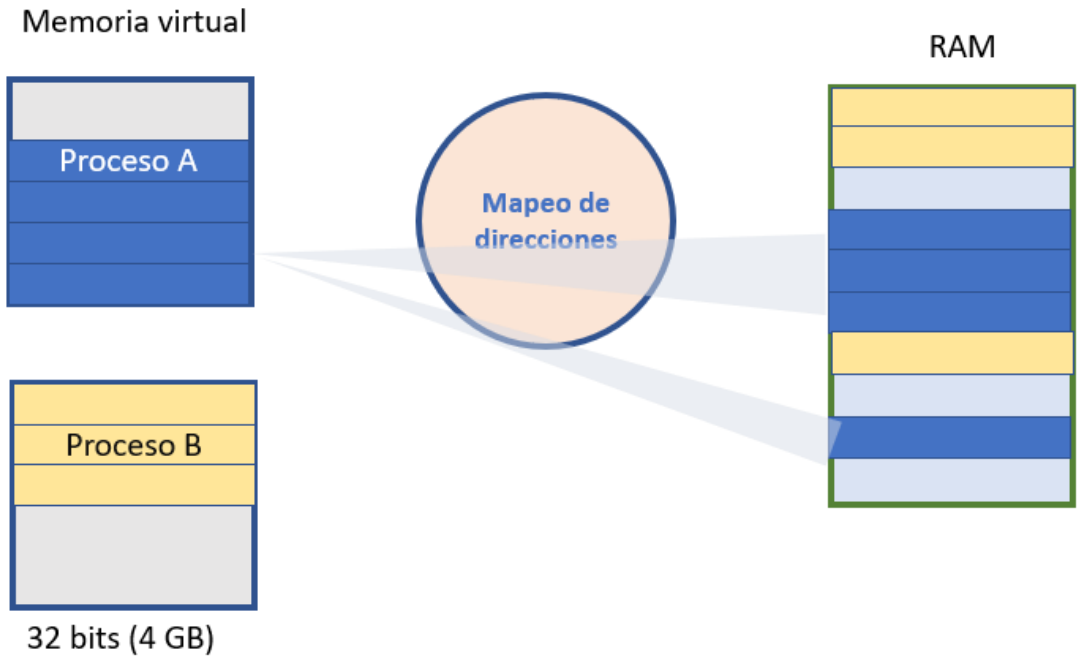
La memoria virtual funciona como un nivel de indirección

Sin memoria virtual Dirección del programa = Dirección en la RAM



Memoria virtual

La memoria virtual con paginación añade flexibilidad al poder asignar espacio físico no contiguo en la RAM. La fragmentación se reduce al mínimo.



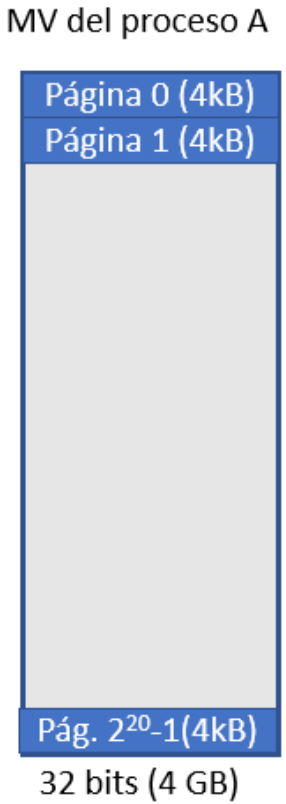
Memoria virtual

Para que la memoria virtual pueda funcionar es necesario que la correspondencia entre página virtual y marco físico se almacene en alguna estructura. Como cada proceso puede ver el mapa de memoria virtual completo los números de página se repiten pero se instancian en marcos diferentes. En consecuencia hay que tener una tabla de correspondencia (tabla de memoria virtual) por cada proceso instanciado y hay que actualizarla cada vez que hay una carga o expulsión.

El kernel mantiene una página por proceso, dentro de su propio espacio de memoria. Como solo puede habar un proceso activo, el kernel trabajará en cada momento con la tabla del proceso en ejecución.

Esta tabla, al igual que e BCP, forma parte de los metadatos de gestión de cada proceso y tiene que crearse cuando este nace y destruirse para liberar espacio al acabar su ejecución.

Memoria virtual

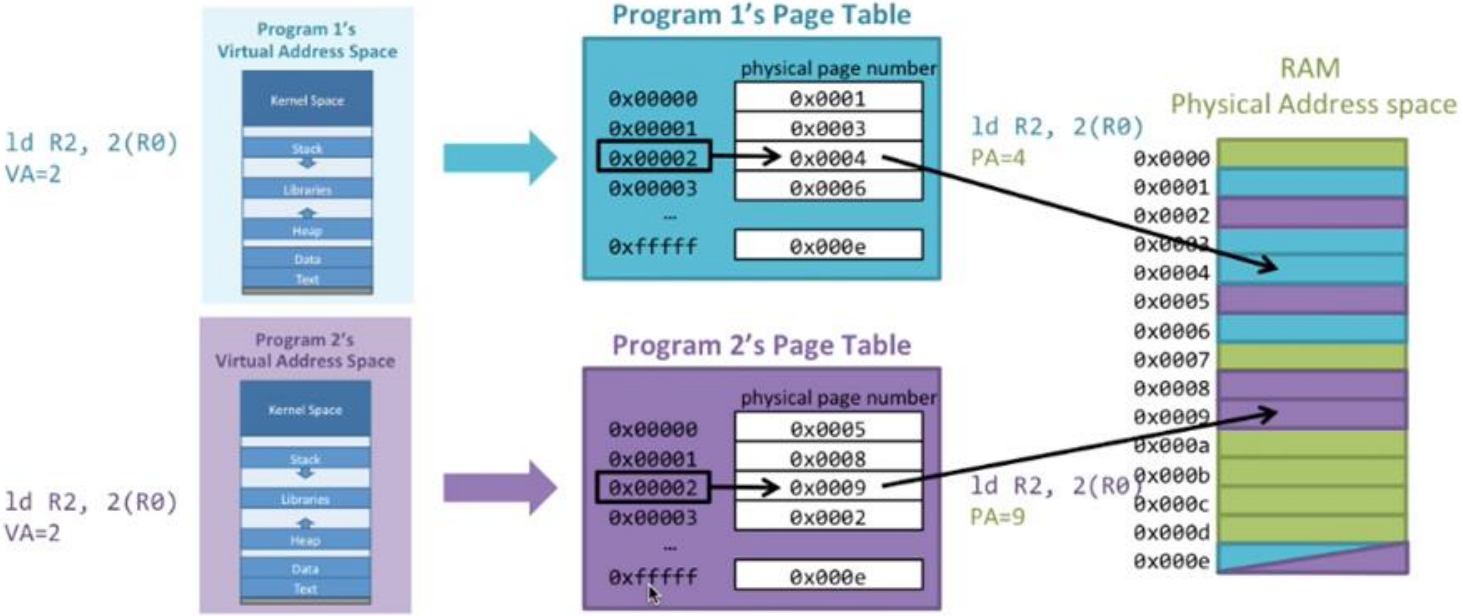


# Página (entrada)	# Marco
0	41
1	9
$2^{20}-1$	5

Tabla de páginas del proceso A de un nivel.
Necesita 2^{20} entradas, una por página (aunque no se esté usando). Se denomina tabla directa. Aprox 1 millón de entradas para 32 bits de direcciones y página de 4kB. Cada fila de la tabla tiene 4 bytes por lo que se ocupan 4 MB por proceso. Esta tabla **tiene que estar cargada completamente en memoria** cuando el proceso está cargado. Si tenemos 100 procesos, sólo las tablas de páginas ocuparían 400 MB

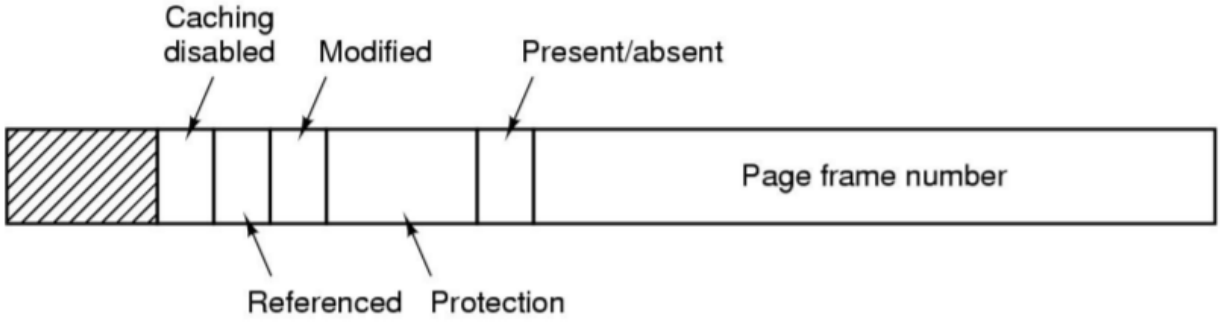
Memoria virtual

Cada proceso tiene su propia tabla de páginas, que reside en el espacio de trabajo del kernel. El SSOO se asegura de que apuntan al mismo marco solo cuando hay compartición legal (por ejemplo, están usando una misma librería dinámica).



Fuente: David Black-Schaffer

Memoria virtual

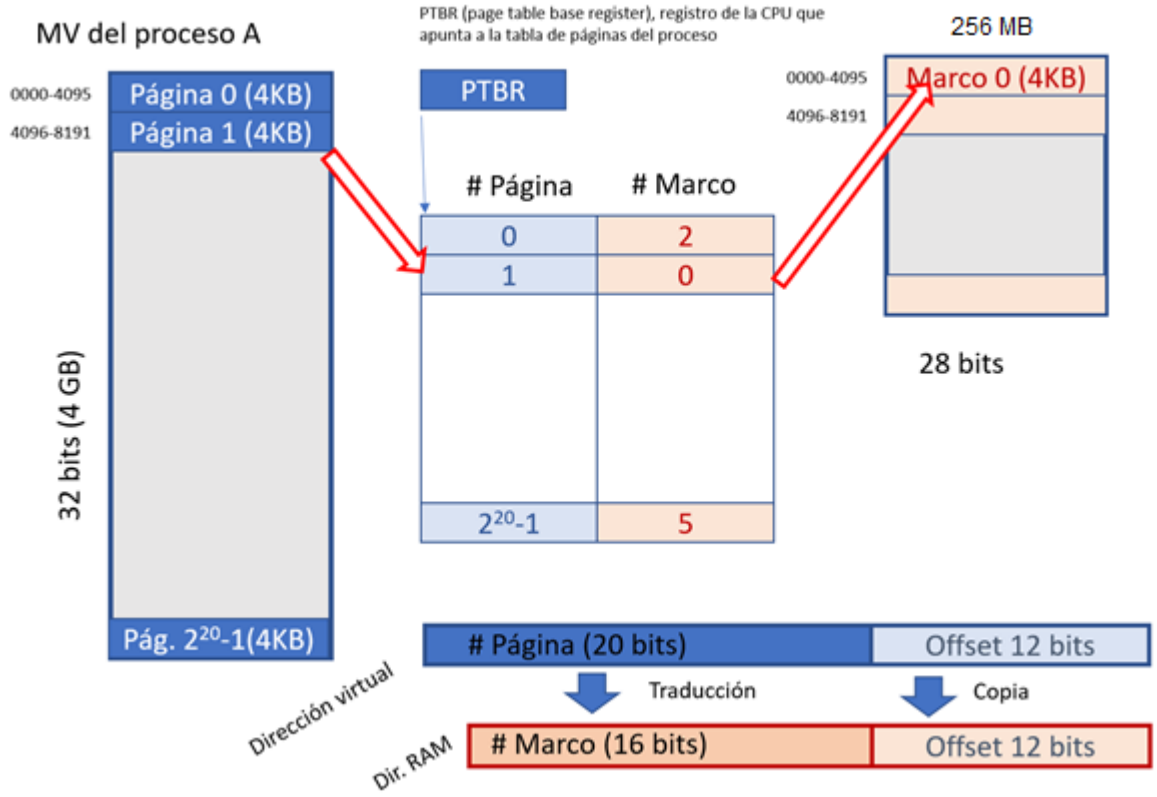


La representación anterior es simplificada. Por cada entrada de la tabla de páginas de un proceso, además del número de marco hay otros bits de control:

- Referenced. Indica que la página ha sido referenciada recientemente y se usa para el algoritmo de sustitución de páginas
- Modified. Indica que los contenidos de la página se han modificado (está “sucia”) y que, por tanto, deberá ser escrita en disco antes de ser expulsada.
- Protection. Indica si la página es de código, se puede escribir, etc.
- Present/absent. Indica si esa entrada es válida. Cuando la página se expulsa, por rapidez, se marca solo ese bit que invalida el resto de informaciones
- Caching disabled. Indica si para acceder a esa página hay que inhabilitar la caché. Se usa para los dispositivos de I/O mapeados en memoria en direcciones físicas fijas.

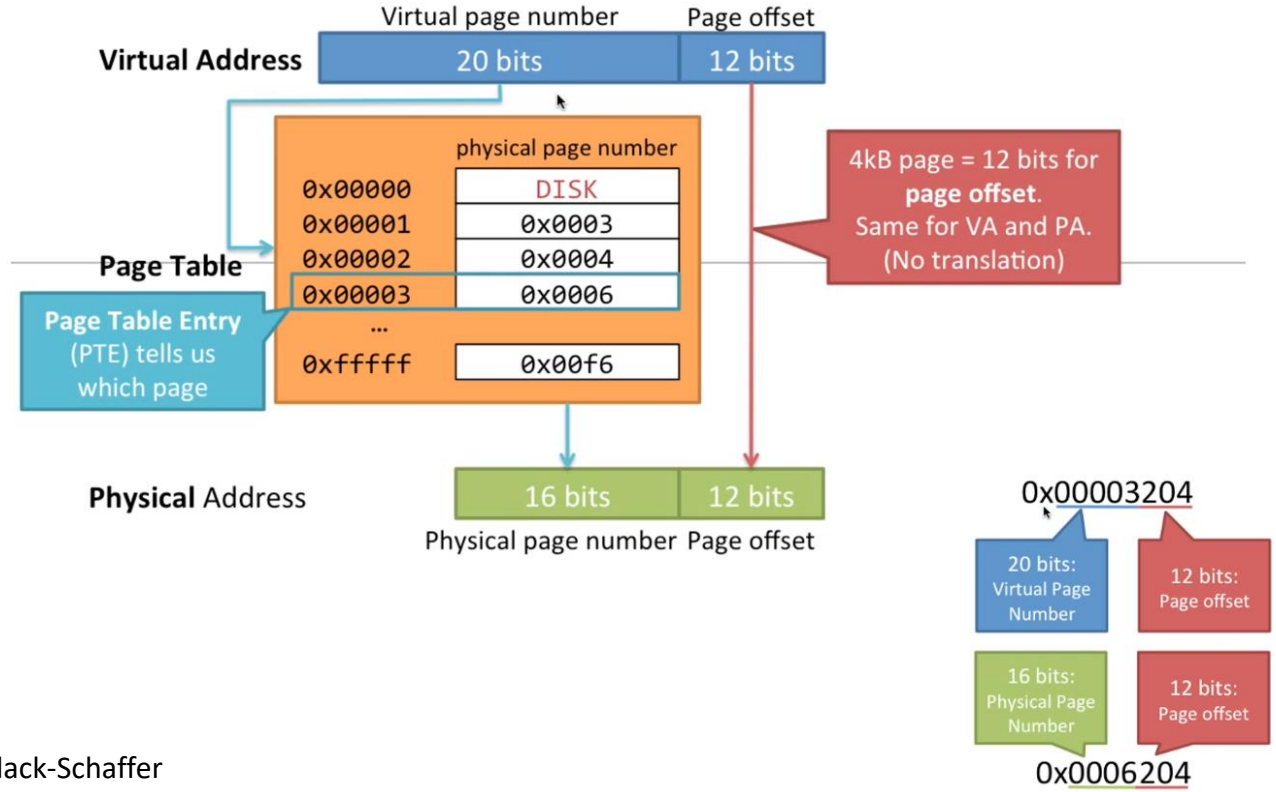
Memoria virtual

¿Cómo se traducen las direcciones?



Memoria virtual

¿Cómo se traducen las direcciones?



Fuente: David Black-Schaffer

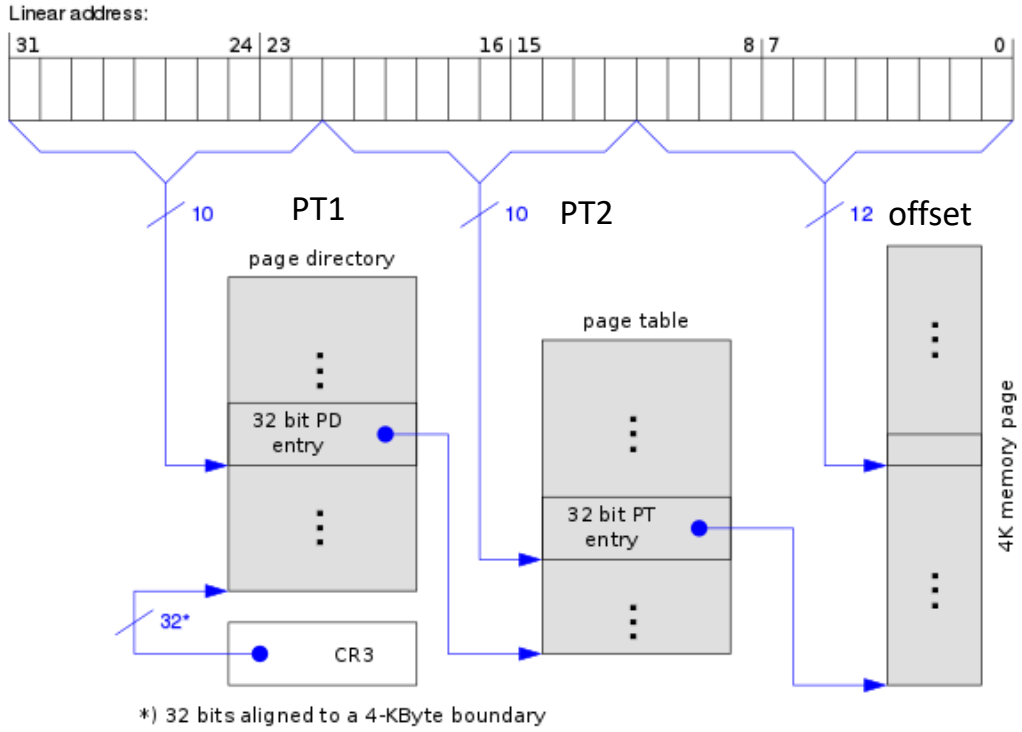
Memoria virtual

Estos ejemplos muestran una de las dificultades de la memoria virtual. Para accede a un dato concreto de la RAM, primero hay que leer una tabla, también cargada en RAM y que alguien se encargue de cambiar número de página por número de marco. Esto no puede funcionar si se hace por software, la solución está en la TLB.

Otro obstáculo muy serio es el tamaño de las tablas. Para una CPU de 32 bits hacen falta 2^{20} entradas (~ 1 millón) en una tabla de único nivel, para una de 64bits nada menos que 2^{52} entradas. Esto es inabarcable, pero afortunadamente la mayoría de las páginas no están instanciadas, por lo que puede recurrirse a una solución ingeniosa para ahorrar mucho espacio de almacenamiento las tablas multinivel.

Memoria virtual

Tablas multinivel



Fuente: By RokerHRO (Own work using: Xfig) [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC BY-SA 3.0

Memoria virtual

Tablas multinivel

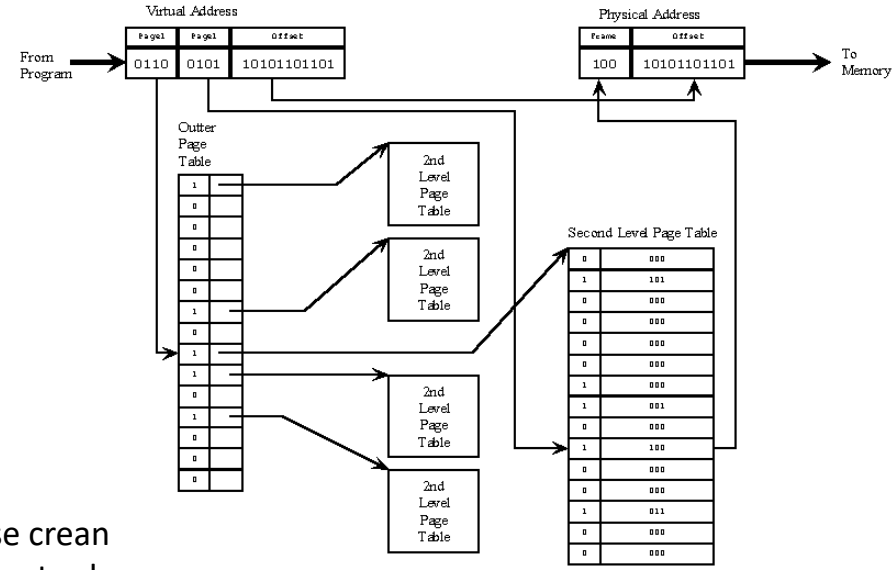
Supongamos que tenemos un sistema de 32 bits con páginas de 4kB. Los 12 bits de menor peso son el offset, permiten referenciar cualquier byte dentro de los 4 kB de la página/marco. Esto es igual que para la tabla de un único nivel.

La diferencia está en los 20 bits de mayor peso. En lugar de tomarlos como un único número que identifica una página dentro de 2^{20}, los separamos en dos grupos (PT1 los de mayor paso, PT2 los de menor). Los bits PT1 son el índice de una tabla única de primer nivel de 1024 entradas. En cada entrada hay un puntero de 32 bits a la dirección de memoria de una hoja de segundo nivel. Nótese que el tamaño de la tabla de primer nivel es 1 página (4 kB). En cada hoja de segundo nivel hay 1024 posibles entradas, cada una de ellas indica un marco de RAM instanciada. Los 10 bits PT2 seleccionan la celda correspondiente de esa tabla de segundo nivel. El ahorro de espacio se consigue porque la mayoría de las páginas no están cargadas, solo hacen falta tantas hojas de segundo nivel como páginas instanciadas/1024.

Memoria virtual

Tablas multinivel

Añaden un nivel adicional de indirección

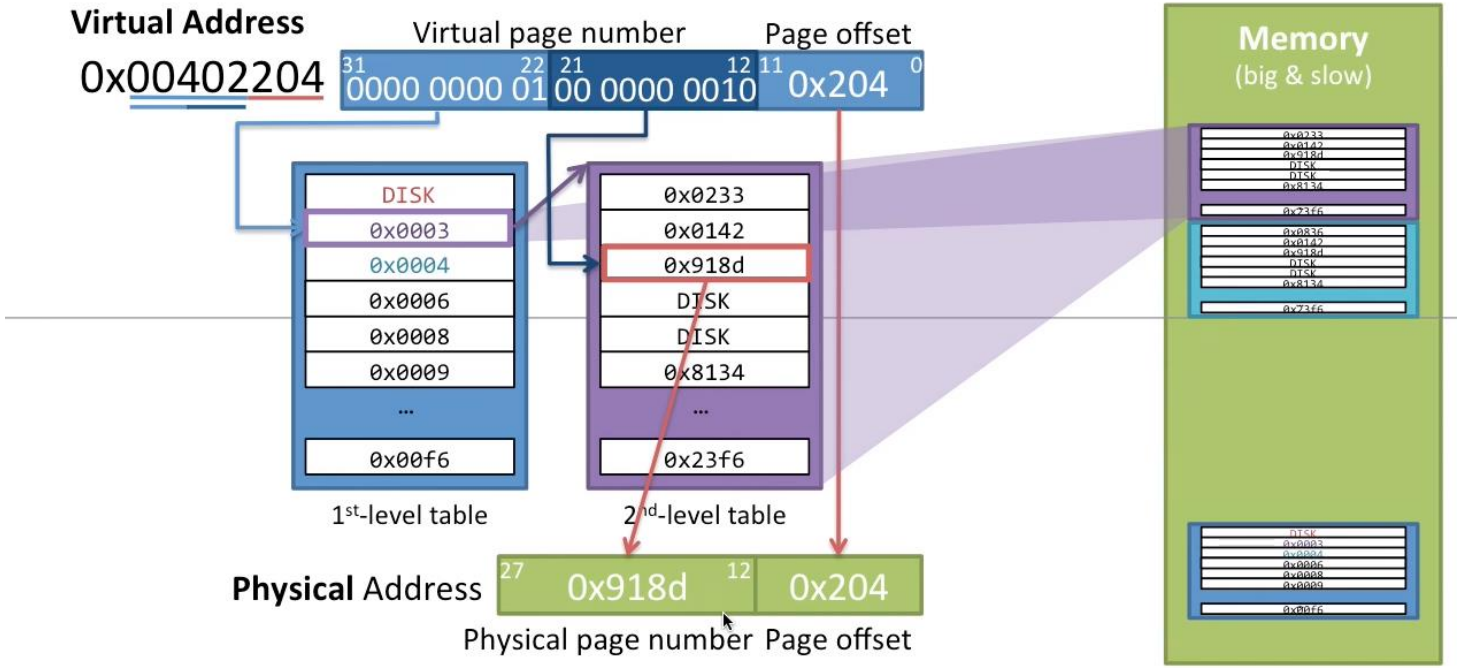


El ahorro se consigue porque solo se crean entradas de segundo nivel para las entradas de primer nivel que están ocupadas. El coste es un acceso adicional a RAM

En este ejemplo solo hay instanciadas 5 hojas de segundo nivel de las 16 posibles.

Memoria virtual

Tablas multinivel

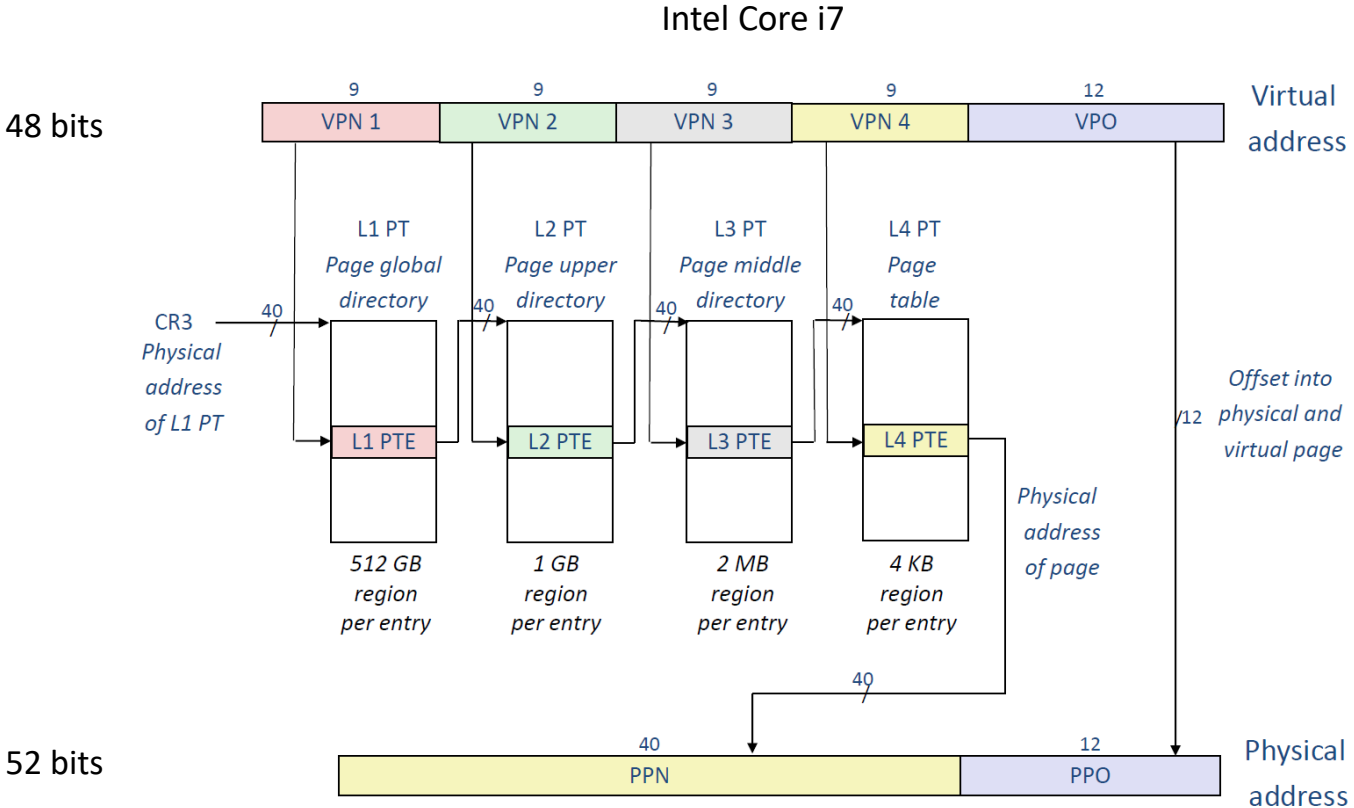


Fuente: David Black-Schaffer

[Two Level Page Table Example (Georgia Tech, Youtube)]

Memoria virtual

Tablas multinivel



Memoria virtual

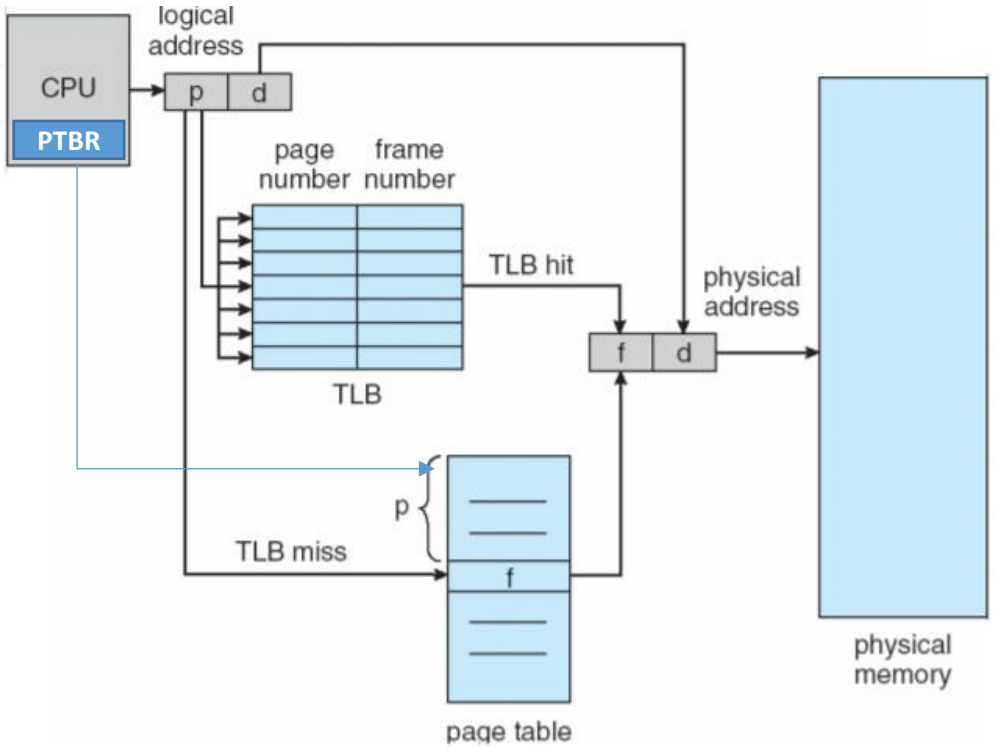
TLB (Translation Lookaside Buffer)

Imaginemos un micro actual con una tabla de 5 niveles. Por cada acceso a memoria del proceso el kernel tendría que realizar 5 accesos solo para saber si la página estaba cargada, y en tal caso realizar la traducción.

Esto no puede realizarse por SW, la MMU dispone de una memoria asociativa especial llamada TLB para llevarlo a cabo. La TLB es pequeña y muy rápida (parecida a una caché) y mantiene la relación entre números de página y marcos accedidos más recientemente. Cada vez que la CPU escribe una dirección virtual, se presenta a la TLB. Como es una memoria asociativa, compara el número de página con todas sus entradas. Si está presente, realiza la traducción de inmediato, sin intervención del Sistema Operativo y el proceso es muy rápido. Si no lo está hay un fallo de TLB. Esto produce una interrupción que despierta al kernel. Si la página está en RAM (fallo de TLB), el kernel toma esos datos de la tabla de memoria virtual del proceso, actualiza la TLB y puede realizarse la traducción. Si no lo está, es un fallo de RAM y hay que cargarla desde disco (fallo de página).

Memoria virtual

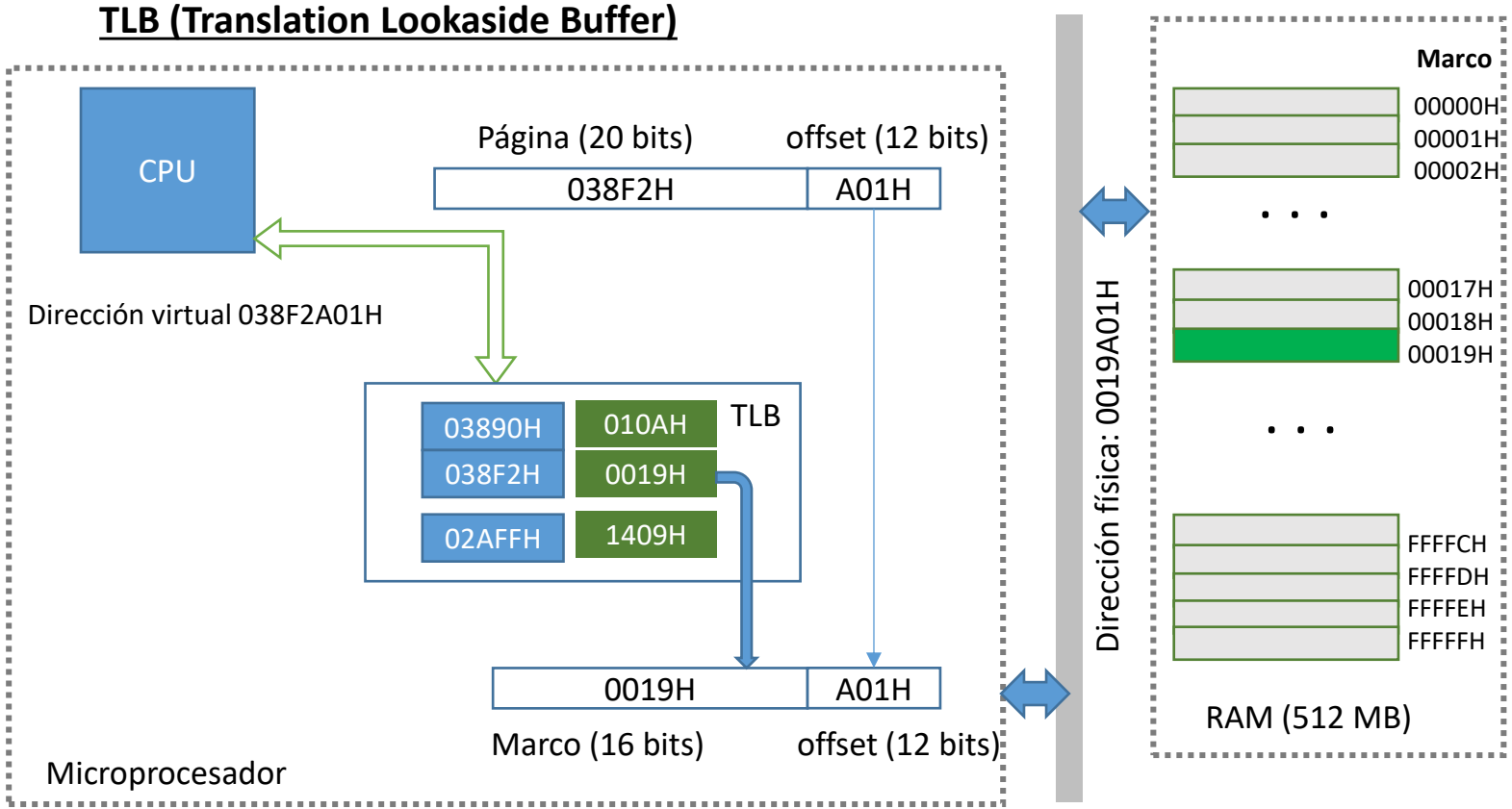
TLB (Translation Lookaside Buffer)



Funcionamiento de la TLB. Si hay acierto el SO no interviene.

La TLB se carga cada vez que hay un cambio de contexto. Un puntero del micro llamado **PTBR** (Page Table Base Register) apunta a la dirección de la tabla del proceso actual en el espacio de memoria del kernel

Memoria virtual



Memoria virtual

TLB (Translation Lookaside Buffer)

- La TLB es parte de la MMU, que incluye también la gestión de la segmentación, la actualización de los bits de estado de la tabla de direcciones (sucia, referenciada), la composición de la dirección de memoria física y la generación de interrupciones por intentos de acceso a direcciones que no son del proceso o en modo no permitido.
- La TLB es muy rápida cuando hay acierto (menos de 1 ciclo de reloj), similar a una caché.
- Una TLB típica tiene 64 entradas (32 para datos y 32 para código), frente al millón de entradas de la tabla de memoria de un proceso.
- La localidad permite que los fallos de TLB sean muy bajos y la memoria virtual se haya consolidado como un sistema casi universal para ordenadores de sobremesa y servidores.
- Cada vez que hay un cambio de contexto, la TLB se tiene que actualizar.
- Durante el arranque del sistema la función de la TLB está anulada porque no existen aun las tablas de memoria. Es lo que se llama **modo real**.

Error común y grave. Suponer que si la tabla es multinivel, la TLB también es multinivel. No es así, es una memoria asociativa que relaciona etiqueta y marco. El dato de la etiqueta (número de página virtual) reside en el último nivel de la tabla de memoria y es lo único que guarda la TLB.

Memoria virtual

TLB + Caché

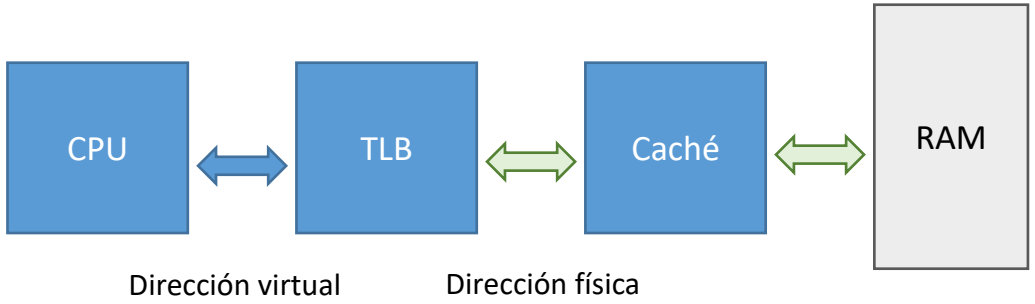
A estas alturas de la lección, puede que confundas la TLB con la caché, la MMU y la tabla de memoria. TLB y caché son memorias asociativas que gestionan zonas muy diferenciadas de la jerarquía de memoria.

¿Sabrías explicar las principales diferencias entre ambas?

Memoria virtual

TLB + Caché

- La TLB traduce direcciones virtuales a direcciones físicas, la caché solo maneja direcciones físicas.
- El Sistema Operativo puede programar la TLB, la caché es inaccesible.
- La TLB se sitúa a la salida de la CPU, la caché entre la TLB y la memoria RAM.
- El fallo de caché no es bloqueante, se resuelve desde el hardware y la CPU no puede ejecutar otra instrucción hasta que termina. Por eso el tamaño de línea es muy pequeño.



Memoria virtual

Fallo de página

El fallo de página se produce cuando la dirección virtual a la que quiere acceder la CPU corresponde con una página no cargada en la memoria. Cuando ocurre esto, el gestor de memoria virtual (parte del kernel) busca un marco libre, se lo asigna a la página en cuestión y lanza una operación de DMA (se verá en la lección 6) para cargar los 4kB desde disco. Como es una operación muy costosa en tiempo, el proceso se bloquea hasta que termine la carga y se cede la CPU a otro.

También puede suceder que esté toda la memoria ocupada. El gestor de memoria tiene que decidir qué página expulsar para dejar un marco libre. Si la página expulsada está sucia habrá que escribirla en disco antes de cargar la nueva. Existen distintos criterios para decidir qué pagina debe expulsarse.

Memoria virtual

Algoritmos de reemplazo de página

- Página no usada recientemente (**NRU**). Se expulsa la página que lleva más tiempo sin usar, para ello se usan los bits M (modificada) y R (referenciada), considerando que la modificación tiene más peso de forma que las páginas con configuración MR = 00 son las primeras que se expulsan (se elige una al azar), luego las 01, 11 y finalmente 11. Los bits M y R se borran periódicamente con la interrupción tick.
- **FIFO**. Se elimina la página que lleva más tiempo en memoria. Para ello el sistema usa una lista unidireccional en la que se borra la página de un extremo y se añade la nueva en el contrario.

(se expulsa) **1** <- 6 <- 12 <- 32 <- 8 <- 46 <- **9** (se añade)

- **Segunda oportunidad**. Una página referenciada gana una "vida" extra. Si la página que se va a expulsar tiene el bit R a 1, se pone a 0 y se coloca en el extremo opuesto de la lista de entrada. [Las páginas con * del ejemplo tienen el bit R a 1]

1* <- 6 <- 12* <- 32 <- 8 <- 46

6 <- 12* <- 32 <- 8 <- 46 <- **1** <- **9**

Memoria virtual

Fallo de página

Ahora podemos ya ver el tratamiento completo del fallo de página:

1. La MMU genera una interrupción que despierta al kernel. Este la atiende, salva los registros del proceso en la pila de usuario e invoca al gestor de memoria.
2. El Sistema Operativo busca en la tabla de gestión de la memoria física si hay un marco libre. Si lo hay, lo asigna a la página, lanza el proceso de carga desde disco y bloquea el proceso. Despertará cuando termine.
3. Si no hay marco libre aplica uno de los algoritmos descritos y decide cuál expulsar. Si la página está sucia (M==1), tiene que escribirla primero a la zona de swap y después cargará la nueva (las dos operaciones son bloqueantes).
4. Cuando ya ha terminado la carga, la instrucción que produjo el fallo se restaura y el proceso puede acceder ya al dato que necesitaba. El proceso queda Ready.
5. Cuando el planificador lo decida el proceso se restaura y sigue con la ejecución en la instrucción que produjo el fallo de página, pero en este caso habrá acierto.

Memoria virtual

Gestión del área de swap

La zona de swap es una partición especial del disco para grabar las páginas expulsadas o cargar desde ella las páginas de un nuevo proceso. En Unix tiene un formateo especial para reducir el sobrecoste de la gestión de ficheros, ya que todas las páginas tienen el mismo tamaño y eso permite tener una estructura plana.

Cuando el ordenador arranca esta zona está vacía. Cuando se crea un proceso se asigna una porción de la zona de swap a sus páginas. En Unix, se guarda una referencia a la zona de swap del proceso en su BCP.

Antes de que el proceso cargue es posible crear su imagen de memoria en la zona de swap e ir cargando desde ella. La otra es cargar solo en memoria e ir escribiendo a swap cuando resulte necesario. La imagen puede cambiar de tamaño a lo largo de la ejecución del proceso, se distinguen distintas zonas en el área de swap para texto, datos y la pila.

Memoria virtual

Gestión del área de swap

Pregunta: ¿Qué pasa si el Sistema Operativo escribe a la zona de *swap* una página con datos confidenciales (claves, números de cuenta bancaria)?

Memoria virtual

Gestión del área de swap

Los que han pensado que los datos están en peligro están completamente en lo cierto. Cuando el Sistema Operativo expulsa una página de datos a la zona de *swap*, no tiene la menor idea sobre cuál es el significado de sus contenidos. Si nos roban el equipo, alguien con el programa adecuado puede volcar los contenidos de la zona de *swap*.

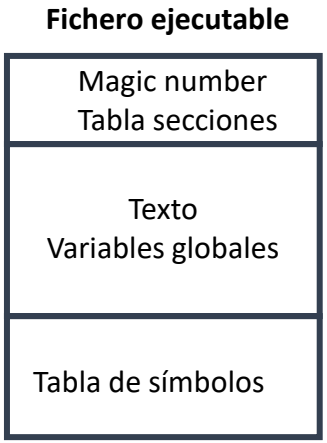
Posibles soluciones:

- Borrar la zona de swap al apagar el equipo. [No se ejecuta si se apaga de forma brusca]
- Cifrar la zona de swap. Tiene consecuencias en el rendimiento global.
- El desarrollador puede marcar determinadas zonas de memoria (variables) como inmunes al swap, se denomina *pinned memory*. En Linux se hace con el servicio `mlock()`.
- Confiar en que no va a pasar (99,999% de los usuarios)

Gestión de la imagen de memoria

Como ya se vio en el tema anterior, la imagen de memoria es el conjunto de la información necesaria para su ejecución, ya sea código máquina o datos. La imagen se crea a partir del ejecutable y este es producto de la cadena de compilación + linkado. El programa se escribe en un lenguaje de alto nival y el compilador crea uno o varios ficheros de objetos. Un fichero objeto consta de:

- Una cabecera que describe cómo se organiza
- Texto: Código máquina y constantes del módulo
- Variables globales
- Información de reubicación
- Tabla de símbolos externos (datos o procedimientos) que no pertecen al módulo, por ejemplo de una librería.
- Información de depuración



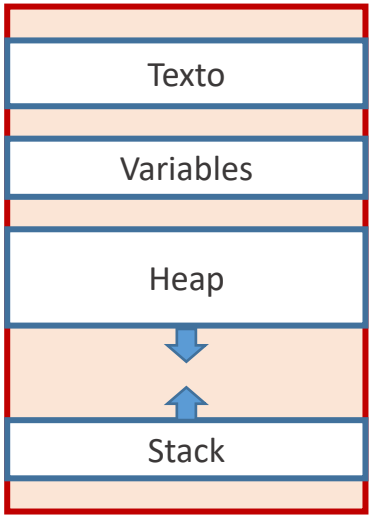


Imagen de memoria

Gestión de la imagen de memoria

Linux maneja el mapa de memoria virtual de cada proceso con una estructura del tipo `mm_struct`. ¿Cuántos campos eres capaz de interpretar?

```
struct mm_struct {
        struct vm_area_struct  *mmap;              /* list of memory areas */
        struct rb_root         mm_rb;              /* red-black tree of VMAs */
        struct vm_area_struct  *mmap_cache;        /* last used memory area */
        unsigned long          free_area_cache;    /* 1st address space hole */
        pgd_t                  *pgd;               /* page global directory */
        atomic_t               mm_users;           /* address space users */
        atomic_t               mm_count;           /* primary usage counter */
        int                    map_count;          /* number of memory areas */
        struct rw_semaphore    mmap_sem;           /* memory area semaphore */
        spinlock_t             page_table_lock;    /* page table lock */
        struct list_head       mmlist;             /* list of all mm_structs */
        unsigned long          start_code;         /* start address of code */
        unsigned long          end_code;           /* final address of code */
        unsigned long          start_data;         /* start address of data */
        unsigned long          end_data;           /* final address of data */
        unsigned long          start_brk;          /* start address of heap */
        unsigned long          brk;                /* final address of heap */
        unsigned long          start_stack;        /* start address of stack */
        unsigned long          arg_start;          /* start of arguments */
        unsigned long          arg_end;            /* end of arguments */
        unsigned long          env_start;          /* start of environment */
        unsigned long          env_end;            /* end of environment */
        unsigned long          rss;                /* pages allocated */
        unsigned long          total_vm;           /* total number of pages */
        unsigned long          locked_vm;          /* number of locked pages */
        unsigned long          saved_auxv[AT_VECTOR_SIZE]; /* saved auxv */
        cpumask_t              cpu_vm_mask;        /* lazy TLB switch mask */
        mm_context_t           context;            /* arch-specific data */
        unsigned long          flags;              /* status flags */
        int                    core_waiters;       /* thread core dump waiters */
        struct core_state      *core_state;        /* core dump support */
        spinlock_t             ioctx_lock;         /* AIO I/O list lock */
        struct hlist_head      ioctx_list;         /* AIO I/O list */
};
```

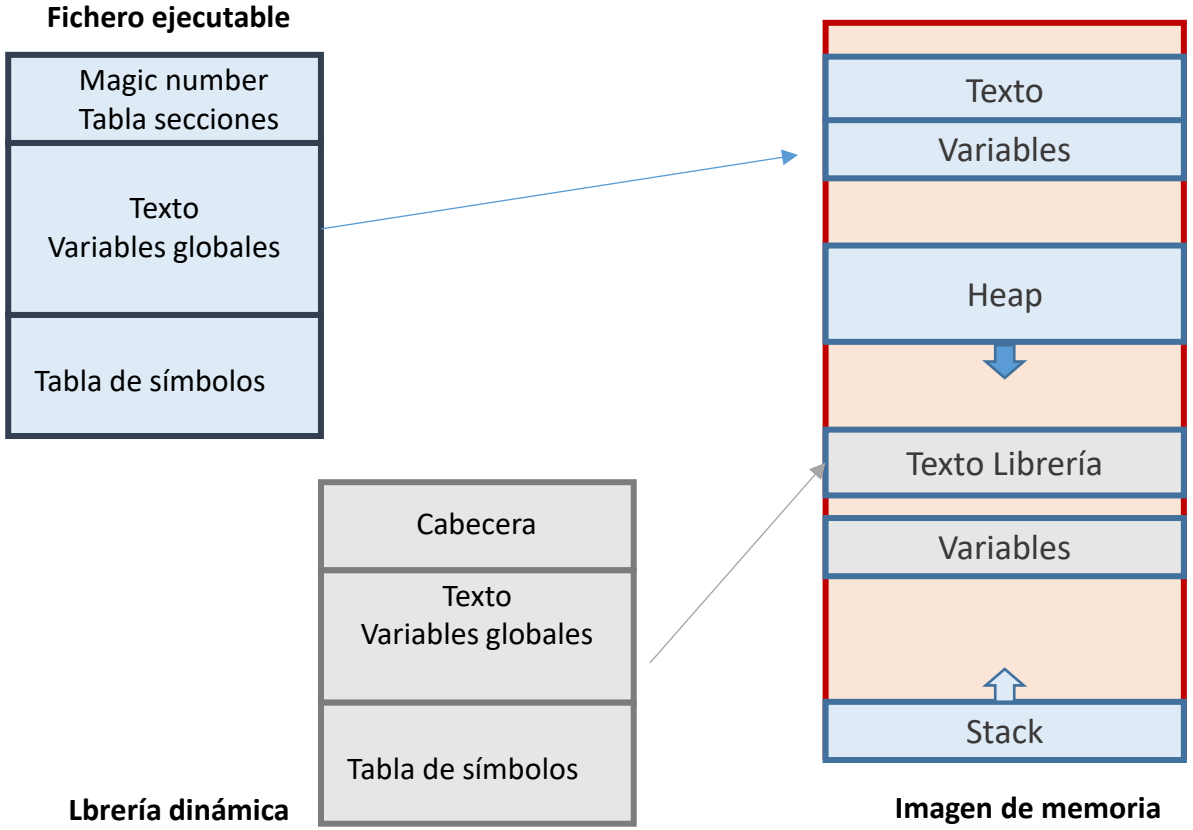
Gestión de la imagen de memoria

Para construir el ejecutable, el linker necesita conocer qué librerías o módulos contienen las referencias externa. En la fase de montaje se reubican estos módulos de acuerdo a la posición final en el ejecutable.

Si el montaje es estático, todo el código de las librerías se incluye en el ejecutable, aunque no se usen las funciones. Tiene el inconveniente de crear ejecutables grandes y de repetir código en las distintas instancias.

Con el montaje dinámico solo se incluye una pequeña rutina capaz de localizar a la librería dinámica e invocarla. La librería se carga la primera vez que un proceso la invoca. Si durante su ejecución un segundo proceso la requiere, simplemente se incluyen esas páginas en su tabla de memoria virtual. El espacio de memoria que ocupa la librería no se libera mientras quede un proceso usándola.

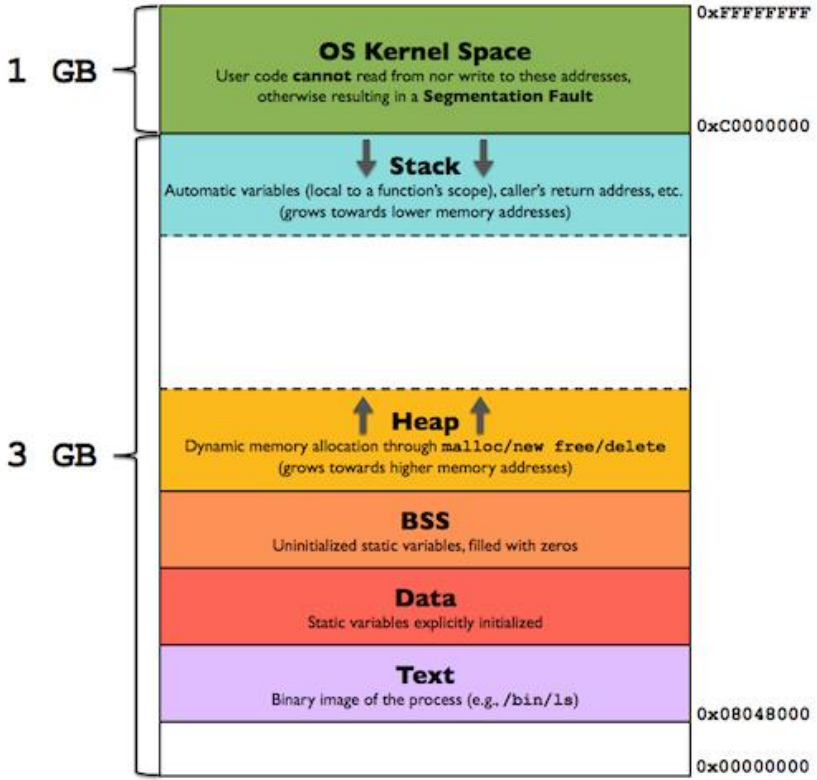
Gestión de la imagen de memoria



Gestión de la imagen de memoria

El kernel está permanentemente cargado en memoria RAM. En todos los procesos se mapea en la tabla de memoria individual de cada proceso. Esto puede parecer que no tiene mucho sentido puesto que el software de usuario no tiene acceso al kernel. Sin embargo, hacerlo así permite evitar cambiar la tabla de memoria virtual cada vez que se produce una interrupción y se pasa de modo usuario a modo supervisor.

En los sistemas de 64 bits, el kernel ocupa la mitad superior de la memoria virtual del proceso.



Gestión de la imagen de memoria

El kernel funciona también con memoria virtual, salvo para llamadas muy específicas que requieran acceso a una dirección física. La razón de este comportamiento es porque si no lo hiciera, cada vez que se pasa de modo usuario a supervisor (o viceversa) habría que anular el funcionamiento de la TLB y pasarla a modo real. Esta operación llevaría mucho tiempo y no aportaría ninguna ventaja.

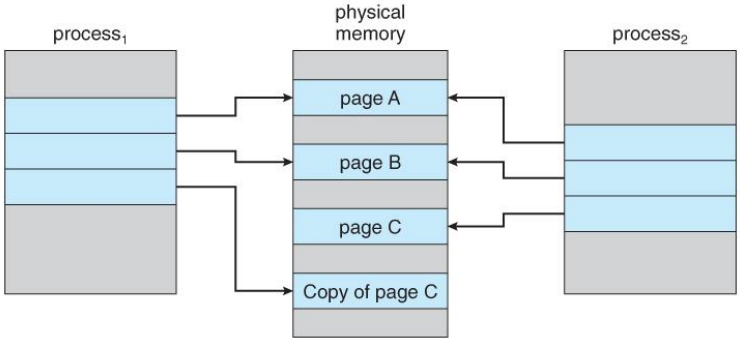
Hay una diferencia importante con los procesos de usuario y es que las páginas del núcleo no son expulsadas a la zona de swap, están cargadas en memoria desde que termina el proceso de arranque de la máquina.

Al ser el propio kernel el encargado de la gestión de la memoria virtual, tiene que construir su propia tabla de mapeo antes de arrancar los procesos en modo usuario.

Gestión de la imagen de memoria

Una técnica muy útil para ahorrar memoria es la llamada **Copy On Write (COW)**. Si tenemos dos instancias del mismo ejecutable, por ejemplo como resultado de un `fork()`, no tiene sentido que se el mismo código máquina se cargue dos veces. Las páginas de memoria virtual del ejecutable se mapean en los mismos marcos.

Esto parece obvio puesto que el ejecutable no cambia, pero ¿qué ocurre con las páginas de datos cuándo se hace un fork? En lugar de duplicarlas se mapean de la misma manera que el código. Si durante la ejecución esos datos se alteran como consecuencia de la actualización de una variable, entonces la página afectada se copia en un marco nuevo, de ahí el nombre de la técnica **Copy on Write**.

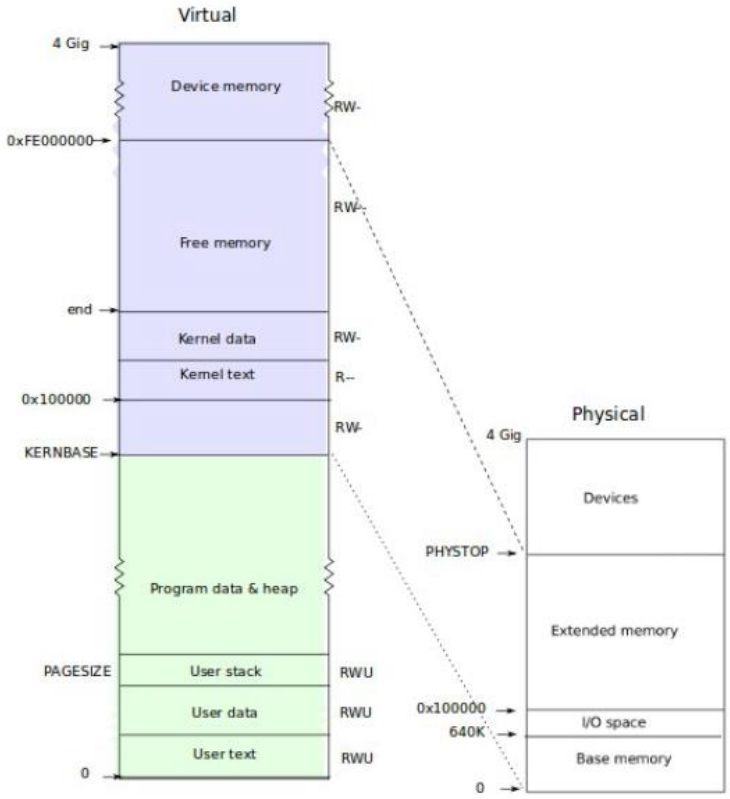


Silberschatz, Galvin & Gagne (2013)

Gestión de memoria en xv6

El núcleo se instancia en todas las instancias de memoria virtual a partir de la dirección KERNBASE. En la RAM, ocupa la zona baja, justo por encima de donde terminaba la zona I/O de la arquitectura del 80286. En modo real, mientras se produce el arranque, es el propio kernel el que traduce de memoria virtual a física simplemente sumando o restando la cantidad KERNBASE, una vez que funciona la TLB ya no tiene que hacerlo.

Los procesos de usuario se instancian en la parte inferior de la memoria virtual.



Gestión de memoria en xv6

Las direcciones de memoria virtual se traducen mediante una tabla de dos niveles y las páginas son de 4 kB.

El registro CR3 (Control Register 3) de la arquitectura x86 es el que se funciona como PTBR (Page Table Base Register).

El sistema xv6 utiliza una lista encadenada de páginas físicas libres para contabilizar qué memoria adicional puede conceder a un proceso durante su ejecución.

Debido a que se trata de una implementación extraordinariamente sencilla, xv6 no dispone de mecanismo de swap a disco.