**PEDECIBA Informática**
**Instituto de Computación – Facultad de Ingeniería**
**Universidad de la República**
**Montevideo, Uruguay**

# Reporte Técnico RT 12-02

# Modelos de memoria en entornos de virtualización

**Mauricio Chimento** **Carlos Luna**
**Gustavo Betarte** **Juan Diego Campo**

**2012**

Modelos de memoria en entornos de virtualización
Chimento, Jesús Mauricio Martín; Luna, Carlos; Betarte, Gustavo; Campo, Juan Diego.
ISSN 0797-6410
Reporte Técnico RT 12-02
PEDECIBA
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, abril  de  2012

# Modelos de Memoria en Entornos de Virtualización

13 de Abril de 2012

Jesús Mauricio Martín Chimento
<jesusmc@fceia.unr.edu.ar>
FCEIA, Universidad Nacional de Rosario, Argentina.

Carlos Luna[a], Gustavo Betarte[b], Juan Diego Campo[c]
<cluna@fing.edu.uy>[a], <gustun@fing.edu.uy>[b],
<jdcampo@fing.edu.uy>[c]
InCo, Facultad de Ingeniería, Universidad de la República, Uruguay.

**Resumen**

La virtualización es una técnica que se utiliza para correr múltiples sistemas operativos en una única máquina física, creando la ilusión de que en realidad cada uno de estos sistemas operativos corre en una máquina virtual diferente. El monitor de máquinas virtuales es el encargado de administrar los recursos compartidos por los sistemas operativos que corren en las máquinas virtuales, de manera que estos sistemas, y las aplicaciones que corren sobre ellos, puedan ejecutar adecuadamente. En particular, el acceso y uso de la memoria principal es un aspecto crítico que el monitor de máquinas virtuales debe controlar.
En este trabajo se presentan los distintos componentes de memoria que están involucrados al virtualizar una computadora y se analizan las maneras en que éstos pueden ser tratados al ser virtualizados.

***Keywords:*** Virtualización, Máquina Virtual, Monitor de Máquinas Virtuales, Hypercall.

# Índice

**1. Introducción** **1**

**2. Memoria Principal** **2**
2.1. Conceptos Básicos . . . . . . . . . . . . . . . . . . . . . . . . 2
2.2. Administración de la Memoria . . . . . . . . . . . . . . . . . . 2
2.3. Unidad de Gestión de Memoria . . . . . . . . . . . . . . . . . 3

**3. Cache** **3**
3.1. Motivación . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3
3.2. Conceptos Básicos . . . . . . . . . . . . . . . . . . . . . . . . 4
3.3. Jerarquía de Memoria . . . . . . . . . . . . . . . . . . . . . . 5
3.4. Diseño de la Cache . . . . . . . . . . . . . . . . . . . . . . . . 6
3.5. Niveles de Cache . . . . . . . . . . . . . . . . . . . . . . . . . 12
3.6. Divisiones de la Cache . . . . . . . . . . . . . . . . . . . . . . 13
3.7. Cache y Memoria Virtual . . . . . . . . . . . . . . . . . . . . 14

**4. TLB** **19**
4.1. Motivación . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
4.2. Conceptos Básicos . . . . . . . . . . . . . . . . . . . . . . . . 20
4.3. Tipos de TLB . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
4.4. Entradas inválidas en la TLB . . . . . . . . . . . . . . . . . . 21

**5. Memoria Principal, Cache y TLB en entornos de Virtualización** **22**
5.1. Virtualización de la memoria . . . . . . . . . . . . . . . . . . 22
5.2. Virtualización de la Cache . . . . . . . . . . . . . . . . . . . . 23
5.3. Virtualización de la TLB . . . . . . . . . . . . . . . . . . . . . 23
5.4. Ejemplos de modelados de Virtualización . . . . . . . . . . . 24
5.5. Técnicas para el manejo de la Memoria Virtualizada . . . . . 26
5.6. Problemas al virtualizar la memoria y la TLB . . . . . . . . . 28

# 1. Introducción

El concepto de ***Virtualización*** surge a principio de los '60 de la mano de IBM. Dicho concepto consiste en abstraer los recursos de un sistema informático, ocultando las características físicas del sistema a usuarios y aplicaciones. De esta manera, IBM buscaba obtener un uso más eficiente de los grandes y costosos mainframes que dicha empresa utilizaba en esa época.

Desde un punto de vista más práctico, la virtualización es una técnica que se utiliza para correr múltiples sistemas operativos, llamados ***guest OSs*** (sistemas operativos invitados), en una sola máquina física, pero creando la ilusión de que en realidad cada uno de los guest OS corre dentro de una ***Máquina Virtual*** (o Virtual Machine, o **VM**) diferente. Dichas **VMs** son una abstracción de la máquina física (i.e. la computadora). Estas son provistas a cada guest OS por una capa delgada de software llamada ***Monitor de Máquina Virtual*** (o Virtual Machine Monitor, o **VMM**). Además, el **VMM** es el encargado de administrar los recursos compartidos por las **VMs** (e.g. Memoria Principal).

Históricamente hay 2 estilos de virtualización: ***Fullvirtualization*** y ***Paravirtualization***. En el primer estilo, cada **VM** es una replica exacta del hardware de la computadora, lo que hace posible que el software y el sistema operativo corran en la **VM** de la misma manera en que lo harían en el hardware de la computadora. En el segundo estilo, cada **VM** es una versión simplificada del hardware de la computadora. Los sistemas operativos que se instalan en las **VMs** (i.e. los guest OSs) son adaptados para que corran adecuadamente. Además, los guest OSs pueden ser divididos en dos clases:

- guest OSs confiables
- guest OSs no confiables

Los guest OSs confiables pueden acceder directamente a las instrucciones privilegiadas (protegidas) y corren en modo supervisor. En cambio, los guest OSs no confiables estos corren en modo usuario. Cuando estos últimos quieran realizar una operación privilegiada (protegida), en su lugar realizarán una ***hypercall***[1] la cual será atendida por el **VMM**. Cabe destacar que las aplicaciones de los guest OSs corren en un nivel de privilegios inferior al de los guest OSs no confiables.

*Organización de este trabajo:* En la sección 2 se presenta el concepto de Memoria Principal y se presentan algunos conceptos claves a tener en cuenta. En la sección 3 se presenta el concepto de Cache, de que manera estas se diseñan y estructuran, como colaboran con el trabajo sobre la Memoria

[1]Una hypercall es para un guest OS lo que una llamada a sistema es para un sistema operativo normal.

Principal y que tipos de Cache hay. En la sección 4 se presenta el concepto de TLB y que tipos de TLB hay. Por último, en la sección 5 se hablará de la Virtualización de los conceptos introducidos en las secciones anteriores, se darán ejemplos de la forma en que algunas de las plataformas de virtualización actuales virtualizan dichos conceptos y de los problemas que pueden surgir al virtualizarlos.

# 2. Memoria Principal

## 2.1. Conceptos Básicos

La Memoria Principal (o memoria **RAM** o, como será mencionada de aquí en más salvo cuando se quiera evitar ambigüedades, simplemente *memoria*) es un arreglo de $2^n$ **palabras** (o bytes) las cuales poseen su propia dirección (única). Además, como puede verse en la figura 1, esta consta de **bloques** de $K$ palabras, siendo 1 el menor tamaño de los anteriores. Por lo tanto, también es válido decir que la Memoria Principal es de tamaño $M = 2^n/K$ bloques. [1]

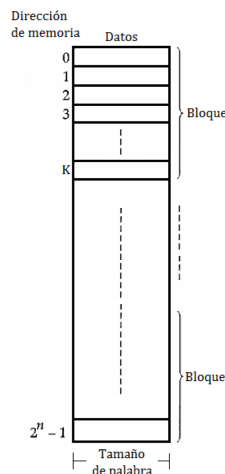


Figura 1: Memoria

## 2.2. Administración de la Memoria

En un sistema monoprocesador sin multiprogramación la memoria se divide en dos partes: una parte para uso del sistema operativo (***espacio de kernel***) y la otra para uso del programa que se encuentre en ejecución (***espacio de usuario***) [1].

A la hora de trabajar con multiprogramación y/o con multitarea, el sistema operativo debe ofrecer alguna forma de administración de la memoria de

tal manera que al tener más de un proceso en ejecución (utilizando el espacio de usuario), estos operen de manera correcta. Las maneras más comunes de administrar la memoria son las siguientes:

- Segmentación de Memoria (o Segmentation)
- Paginación de Memoria (o Paging)

#### Segmentación de Memoria

En Segmentación de Memoria, o simplemente ***Segmentación***, cada proceso se divide en una serie de segmentos los cuales, a la hora de cargar un proceso, son situados en particiones de la memoria. Dichas particiones pueden ser creadas dinámicamente (tamaño variable) o estáticamente (tamaño fijo) [1]. Además, para que un proceso pueda ejecutarse todos sus segmentos deben estar cargados en la memoria (en particiones no necesariamente contiguas).

#### Paginación de Memoria

En Paginación de Memoria, o simplemente ***Paginación***, la memoria se divide en una serie de marcos, todos del mismo tamaño, y cada proceso se divide en una serie de páginas del mismo tamaño que los marcos. Luego, un proceso se carga en memoria situando todas sus páginas en marcos libres de la memoria, los cuales no necesariamente deben ser contiguos.

### 2.3. Unidad de Gestión de Memoria

La Unidad de Gestión de Memoria (o **MMU** por sus siglas en inglés) es un componente de hardware responsable de manejar los accesos a la memoria pedidos por el micro. Además, esta es la encargada de traducir una dirección virtual en una dirección física, de realizar tareas de protección de memoria y del control de la Cache, entre otras cosas.

## 3. Cache

### 3.1. Motivación

Pese al amplío avance de la tecnología, el incremento de velocidad de los micros no se ha dado de manera equitativa al incremento de la velocidad de las memorias. Los micros son muchos más rápidos que las memorias, lo que hace que estas últimas no sean lo suficientemente rápidas para almacenar y transmitir los datos que los primeros mencionados necesitan, generando retardos considerables dado que los micros tienen que esperar a que las memorias estén disponibles para poder trabajar.

Una posible solución al problema de rendimiento antes mencionado radica en no utilizar un único componente de memoria sino en utilizar una memoria lenta de gran capacidad y una memoria de poca capacidad pero con tiempos de acceso rápidos, haciéndolo de la siguiente manera: la memoria de gran capacidad será la Memoria Principal y la otra memoria, que recibirá el nombre de Memoria Cache, o simplemente Cache, estará ubicada entre la Memoria Principal y el micro, siendo la Cache la clave para equiparar la velocidad de la Memoria Principal con la velocidad del micro.

## 3.2. Conceptos Básicos

Como fue mencionado en la sección anterior, y ahora se ilustra en la figura 2, la Cache se halla ubicada entre la Memoria Principal y el micro. Además, también se mencionó que la Cache es la pieza clave en la solución del problema de rendimiento. Ahora, ¿por qué esto es así?

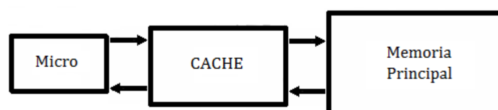


Figura 2: Ubicación de la Cache

Esto es así porque la Cache hará las veces de cache de memoria para la memoria, de ahí su nombre. En ciencias de la computación, una cache es un conjunto de datos los cuales han sido copiados de otros datos originales y tienen la propiedad de que el costo de acceder a los mismos, respecto del tiempo, es menor que el costo de acceder a los datos originales. Además, cuando se accede por primera vez a un dato, se hace una copia en la cache. De esta manera los accesos siguientes se realizan a dicha copia, haciendo que el tiempo de acceso medio al dato sea menor.

La Cache contendrá una copia de parte de la memoria. Cada una sus entradas está compuesta por un ***índice*** que identifica a que sección de la Cache se está haciendo referencia (hay $S$ secciones, donde $S << 2^n$), un ***bloque*** de la memoria y una ***etiqueta*** que identifica la dirección de dicho bloque [1].

Cuando el procesador intente acceder a un dato de la memoria, se comprobará si el mismo está en la Cache. De ser así, se dice que se ha producido un ***Cache hit*** y el dato será enviado al micro. Caso contrario, se dice que se ha producido un ***Cache miss*** y se agrega a la Cache el bloque de la memoria que contenga al dato en cuestión. Como la Cache tiene poca capacidad, los datos pueden tener que ser desalojados de la Cache con el fin de agregar nuevos datos a la misma. Además, debido al fenómeno de la ***cercanía de referencias***[2], cuando se agregue a la Cache un bloque de datos

[2]En un momento concreto, los programas acceden a una parte relativamente pequeña

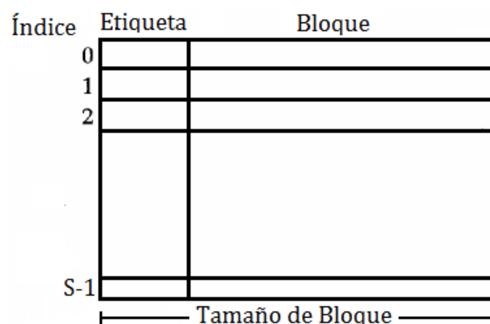


Figura 3: Cache

para satisfacer una referencia a memoria, si dicho bloque había sido alojado previamente en la Cache entonces es probable que ya se hayan hecho antes otras referencias a datos del mismo bloque [1].

## 3.3. Jerarquía de Memoria

Al agregarse la Cache, la computadora pasa a no depender más de un único componente de memoria. Lo anterior sumado a la manera en que se utiliza la Cache hace que la computadora ahora pase a tener una ***Jerarquía de Memoria*** [1]. Dicha jerarquía se divide tradicionalmente como se muestra en la figura 4.

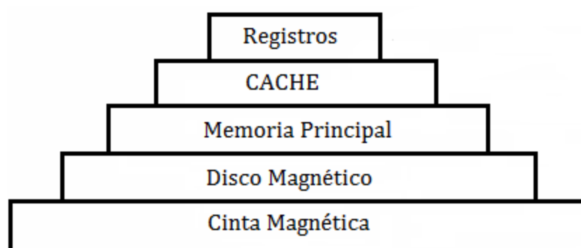


Figura 4: Jerarquía tradicional de la memoria

En esta jerarquía, a medida que se desciende por la pirámide, se cumplen las siguientes condiciones [1]:

- Disminución del coste por bit.
- Aumento de la capacidad.
- Aumento del tiempo de acceso.
- Disminución de la frecuencia de acceso a la memoria por parte del micro.

de su espacio de direcciones. Dichas direcciones tienden a estar agrupadas.

## 3.4. Diseño de la Cache

A la hora de diseñar una Cache hay que tener en cuenta los siguientes aspectos [1]:

- Tamaño de la Cache.
- Función de correspondencia.
- Algoritmo de reemplazo.
- Política de escritura.

### Tamaño de la Cache

Como ya se dijo anteriormente, el tamaño de la Cache viene limitado por el coste de este tipo de memorias, por lo que estas suelen ser de poca capacidad. Esto hace que los tiempos de acceso a la misma sean rápidos. Además, a menor capacidad de Cache, mayor velocidad de acceso.

### Función de Correspondencia

Cuando se agrega a la Cache un nuevo bloque de datos, la función de correspondencia es la encargada de determinar la posición en la cual este se ubicará. Hay que tener en cuenta que para agregar dicho bloque otro bloque que ya resida en la Cache puede llegar a tener que ser reemplazado. Esto hace que sea conveniente reducir la probabilidad de reemplazar un bloque que se vaya a necesitar en un futuro cercano [1].

Los tipos de correspondencias usuales son los siguientes:

- Directa
- Asociativa
- Asociativa por Conjuntos

**Directa**

En la correspondencia Directa al bloque $i$-ésimo de la memoria le corresponde siempre el índice ***i módulo S*** en la Cache, donde $S$ es la cantidad de secciones de la Cache e $i$ es la dirección del bloque. A la hora de acceder a la Cache, las direcciones de la memoria son vistas como se las muestra en la figura 5 [2]:

Dirección de Mem. Ppal.

| TAG | IND | Desplazamiento |
|---|---|---|

Figura 5: Dirección de Memoria en Correspondencia Directa.

, donde **TAG** corresponde a la etiqueta del bloque al que pertenece la dirección, **IND** corresponde al índice de la Cache en la que debe encontrarse dicho bloque y **Desplazamiento** corresponde al lugar en el bloque donde se ubica dicha dirección.

Obviamente, como puede apreciarse en el ejemplo de la figura 6, una sección de la Cache estará asociada a más de un bloque. Por lo tanto, solo se producirá un Cache hit cuando la etiqueta del bloque situado en la sección *IND* de la Cache sea igual a *TAG*.

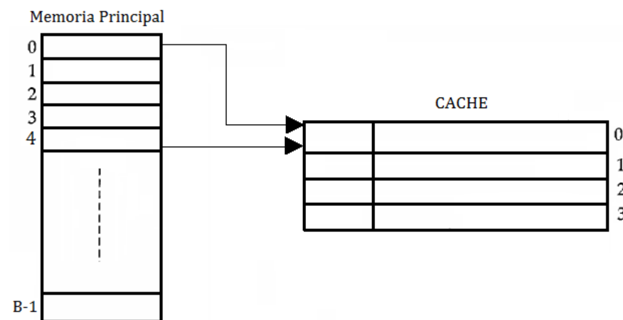


Figura 6: Ejemplo de uso de Correspondencia Directa con una Cache de 4 secciones y B bloques de memoria.

**Asociativa**

En la correspondencia Asociativa cualquier bloque de la memoria puede ser colocado en cualquiera de las $S$ secciones de la Cache. A la hora de acceder a la Cache, las direcciones de la memoria son vistas como se las muestra en la figura 7 [2]:

Dirección de Mem. Ppal.

| TAG | Desplazamiento |
|---|---|

Figura 7: Dirección de Memoria en Correspondencia Asociativa.

, donde **TAG** corresponde a la etiqueta del bloque al que pertenece la dirección y **Desplazamiento** corresponde al lugar en el bloque donde se ubica dicha dirección.

Al inspeccionarse la Cache en busca de un bloque de la memoria, $TAG$ será comparado con las etiquetas de los anteriores. Si ocurre una coincidencia

entonces se habrá producido un Cache hit.

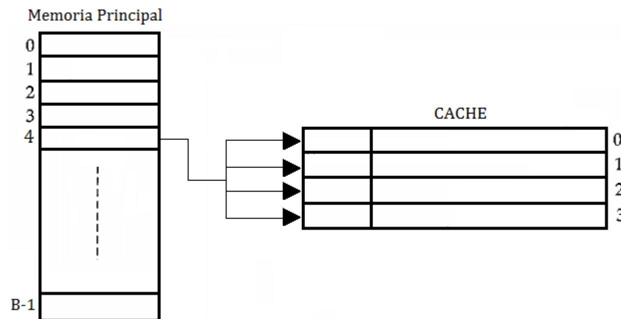


Figura 8: Ejemplo de uso de Correspondencia Asociativa con una Cache de 4 secciones y B bloques de memoria.

**Asociativa por Conjuntos**

En la correspondencia Asociativa por Conjuntos la Cache se divide en $k$ conjuntos de bloques. Luego, al bloque $i$-ésimo de memoria le corresponde el conjunto ***i módulo k***. Dicho bloque podrá ubicarse en cualquier posición de ese conjunto.

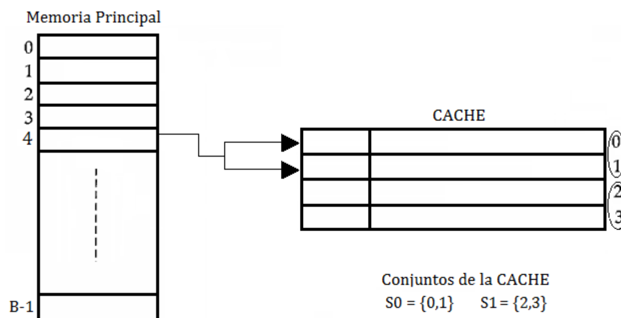


Figura 9: Ejemplo de uso de Correspondencia Asociativa por Conjuntos con una Cache de 4 secciones y B bloques de memoria.

A la hora de acceder a la Cache, las direcciones de la memoria son vistas como se las muestra en la figura 9 [2]:

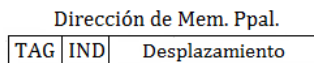


Figura 10: Dirección de Memoria en Correspondencia Asociativa por Conjuntos.

, donde **TAG** corresponde a la etiqueta del bloque al que pertenece la direc-

ción, **IND** corresponde al conjunto de la Cache en la que debe encontrarse dicho bloque y **Desplazamiento** corresponde al lugar en el bloque donde se ubica dicha dirección.

Al inspeccionarse la Cache en busca de un bloque de la memoria, *TAG* será comparado con las etiquetas de los bloques que residan en el conjunto *IND*. Si ocurre una coincidencia entonces se habrá producido un Cache hit.

### Algoritmo de Reemplazo

Cuando se utiliza Correspondencia Directa para agregar un bloque de la memoria a la Cache, si en el lugar donde este debe ser colocado ya se halla ubicado otro bloque entonces simplemente se libera en la Cache el lugar en cuestión y se coloca en el mismo el nuevo bloque. En cambio, si se utiliza alguno de los otros tipos de correspondencia se necesitará de algún algoritmo que indique de que manera debe escogerse un bloque saliente a la hora de agregar un bloque a la Cache, siempre y cuando no haya un lugar libre en la Cache para el bloque entrante claro esta. Dicho algoritmo suele ser llamado Algoritmo de Reemplazo. Los Algoritmos de Reemplazo deben ser implementados en hardware y deben ejecutarse con mucha rapidez para no influir de manera negativa en la performance de la computadora [3]. Además, suele ser conveniente reemplazar aquel bloque que tenga menos probabilidad de ser utilizado en un futuro cercano [1]. Entre los antes mencionados, los algoritmos tradicionalmente más usados son:

- FIFO.
- RAND.
- LFU.
- LRU.

#### FIFO

Este algoritmo recibe su nombre por sus siglas en inglés, ***First In First Out*** (Primero en Entrar Primero en Salir). El mismo consiste en asociar a cada bloque de datos la hora en la que han sido alojados en la Cache. Luego, cuando un bloque de datos en la anterior deba ser reemplazado, se elegirá al más viejo [4]. En otras palabras, y como su nombre lo indica, el primer bloque de datos en entrar a la Cache será el primer bloque de datos en salir de la misma.

Cabe destacar que cuando los bloques de datos son páginas, i.e. utilizando paginación, si el tamaño de la memoria utilizada (en nuestro caso la Cache) aumenta también puede aumentar la velocidad de ocurrencia de los *page-faults*. Esto se conoce como la ***anomalía de Belady*** [5]. Además, por su manera de reemplazo, este algoritmo puede quitar de la Cache un bloque

de datos que este siendo muy utilizado y conservar en la misma uno de poca utilización.

### RAND

Este algoritmo recibe su nombre al cortar su nombre en inglés, ***Random*** (Aleatorio). El mismo consiste simplemente en elegir el bloque de datos a reemplazar en la Cache de manera aleatoria.

### LFU

Este algoritmo recibe su nombre por sus siglas en inglés, ***Least Frequently Used***. El mismo consiste en asociar un contador diferente a cada bloque de datos. Dichos contadores se incrementarán cada vez que se haga referencia a su bloque de datos asociado. Luego, cuando un bloque de datos en la Cache deba ser reemplazado, se elegirá el bloque al que se hallan hecho menos referencias (i.e. que tenga el contador más bajo) [4]. La manera en la que trabaja este algoritmo presenta a priori 2 problemas:

- Si durante la fase inicial de un proceso se realizan múltiples referencias a un bloque de datos, al que llamaremos $A$, entonces dicho bloque tendrá un contador elevado que probablemente será mayor al de los otros bloques. Ahora bien, si en el futuro cercano no vuelven a hacerse referencias a $A$ esto ocasionará que este permanezca en la Cache innecesariamente. Esto ocurre dado que cuando se deba realizar un reemplazo de un bloque de datos en la Cache, el bloque elegido será uno de menor contador que el de $A$, pero que en realidad tendrá más chances de volver a ser utilizado antes que el anterior.

  Una posible solución al problema antes mencionado puede ser realizar en intervalos regulares de tiempo un corrimiento de un bit a la derecha a cada contador, lo que reduce sus valores en una escala temporal exponencial [4].

- Puede darse el caso en el que la frecuencia en la que se hace referencia a cada bloque de datos en la Cache sea la misma. Esto significa que a la hora de que un bloque de datos deba ser reemplazado de la Cache todos tendrán el mismo valor de contador. Por lo tanto, esto hace que deba implementarse alguna política de elección para romper estos "*empates*". Dicha política suele ser aplicar el algoritmo LRU (o reglas similares a la forma de trabajo del anterior).

### LRU

Este algoritmo recibe su nombre por sus siglas en inglés, ***Last Recently Used***. El mismo consiste en asociar a cada bloque de datos la hora en la cual han sido utilizados por última vez. Luego, a la hora de realizarse el

reemplazo de un bloque de datos en la Cache, se elige el bloque que hace más tiempo que no ha sido utilizado [4].

Una dificultad que puede presentar este algoritmo es que para identificar el bloque que hace más tiempo que no ha sido utilizado son necesarios mecanismos de hardware, los cuales deben ser de rápida ejecución [1].

#### Comparación

Los Algoritmos de Reemplazo pueden dividirse en 2 categorías [3]:

- Algoritmos basados en el uso de los bloques (o ***Usage-based algorithms***).
- Algoritmos no basados en el uso de los bloques (o ***Non-Usage-based algorithms***).

La diferencia entre las anteriores es que en la primera categoría los algoritmos tienen en cuenta cuando son utilizados los bloques de datos para operar mientras que en la segunda categoría no. Es fácil deducir a partir de la manera en que los algoritmos antes mencionado trabajan que FIFO y RAND pertenecen a la segunda categoría mientras que LFU y LRU a la primera. Ahora bien, dentro de estas categorías pueden ser destacados como los algoritmos más representativos de las mismas a FIFO y LRU. El primero por su fácil implementación y el segundo por su proximidad a un Algoritmo de Reemplazo óptimo [4]. Por último, es destacable que el algoritmo LRU realiza un 12 por ciento menos de reemplazos que FIFO, por lo que en cuestiones de performance el primero es mejor que el segundo. En la sección 2.4 de [3] puede verse una tabla comparativa entre LRU y FIFO que justifica dicho porcentaje.

### Política de Escritura

Si el contenido de un bloque que se encuentra en la Cache es modificado, entonces dicho bloque debe ser escrito nuevamente en la memoria (particularmente antes de ser reemplazado). La política de escritura dicta cuando tiene lugar la operación de escribir en memoria [1]. Existen (por lo menos) 2 técnicas para esto:

- Escritura Inmediata (o Write-through).
- Escritura Demorada (o Copy-back o Write-Back).

#### Escritura Inmediata

Cuando una operación de escritura se lleva acabo entonces la información escrita en la Cache se transmite inmediatamente a la memoria. Si se utiliza esta técnica, entonces la memoria siempre contendrá una copia *up-to-date*

(actualizada) de toda la información del sistema y una copia válida del total de sus estados en cualquier momento dado. Esto permite que si el micro falla, el sistema pueda restaurarse de manera más sencilla. [3]

#### Escritura Demorada

Cuando una operación de escritura se lleva acabo entonces solo se modifica la Cache. Dicho cambio será transmitido a la memoria más tarde, cuando ocurra un *Cache miss*. Esta técnica puede complicar la lógica de la Cache. Ahora se necesitará un bit especial, llamado ***dirty bit***, para saber cuando se debe copiar información desde la Cache a la memoria. [3]

#### Comparación

La Escritura Demorada por lo general tiene menos tráfico a memoria que la Escritura Inmediata dado que la primera solo necesita acceder a la memoria cuando ocurra un *Cache miss* mientras que la segunda debe hacerlo en cada modificación de la memoria. Además, ambas técnicas necesitan utilizar un buffer, aunque con fines diferentes. La Escritura Demorada lo utiliza para que la información a copiarse en la memoria pueda ser manejada de manera temporal sin interferir en las búsquedas en la Cache [3]. La Escritura Inmediata lo utiliza para almacenar información a ser copiada en la memoria con el objetivo de que el sistema no tenga que esperar a que la copia de dicha información se halla completado para poder trabajar [3].

## 3.5. Niveles de Cache

En lo que va de este trabajo siempre se hizo referencia a la Cache como una sola memoria. Ahora bien, esto no es necesariamente así. La Cache puede dividirse jerárquicamente en al menos 3 niveles, lo que modificaría nuestra visión de la jerarquía tradicional de memoria como lo muestra la figura 11 (manteniendo cada una de estas lo mencionado en la sección 2.2).

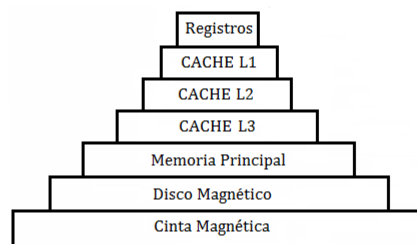


Figura 11: Jerarquía de la Cache

**L1:** Conocida como Cache interna, esta se encuentra en el nivel más cercano al micro, con lo que el acceso se produce a la velocidad de trabajo del

anterior (la máxima velocidad). Además, presenta un tamaño muy reducido.

**L2:** Conocida como Cache externa, inicialmente se instalaba en la placa base (en el exterior de la CPU). Los tamaños típicos de esta Cache oscilan en la actualidad entre 256 KB y 1 MB.

**L3:** Se encuentra en algunas placas base, micros y tarjetas de interfaz.

## 3.6. Divisiones de la Cache

El tiempo de acceso a la Cache y su performance pueden ser mejorados dividiendo la Cache en dos partes. Una parte para los datos y otra parte para las instrucciones. Esto duplica el tiempo de acceso a Cache dado que ahora pueden atenderse dos peticiones en el tiempo en que antes se atendía una sola [3]. Por ejemplo, una instrucción puede ser leída y una carga o almacenamiento de información pueden ser realizadas al mismo tiempo [6]. Dicha división suele ser realizada al utilizar niveles de Cache y, como se muestra en [7], suele ser hecha en la Cache L1 (Cache Interna). Además, las partes generadas a partir de esta división reciben el nombre de:

- Cache de Instrucciones (o Instruction Cache).
- Cache de Información (o Data Cache).

Cabe destacar que esta división siempre ha sido utilizada en computadoras con micros de arquitectura MIPS [6].

**Cache de Instrucciones**
La Cache de Instrucciones se utiliza para alojar instrucciones. De esta manera se busca aumentar la velocidad de búsqueda de las mismas en memoria. Además, esta Cache puede contener otras cosas, como por ejemplo información de *branch prediction*, y también puede realizar algunas operaciones limitadas. [7]

**Cache de Información**
Una Cache de Información se utiliza para alojar información de las aplicaciones. Antes de que el micro pueda operar sobre la información, esta debe ser cargada desde la memoria a la Cache de Información. Luego, el elemento necesario es cargado desde la Cache de Información a un registro para que la instrucción que tenga que utilizar este valor pueda operar. [7]

### Problemas introducidos al dividir la Cache [3]

Dividir la Cache como fue mencionado anteriormente introduce algunos problemas. La consistencia de la información en memoria es uno de ellos. Al dividirse la Cache, ahora existen dos copias de la información mientras que antes existía una sola. Otro problema es que esta división puede ocasionar un uso ineficiente de la Cache dado que como la cantidad información o de instrucciones necesarias para que una aplicación trabaje varía de programa en programa, si estas no son almacenadas conjuntamente entonces deberán existir cada una dentro de su propia memoria, siendo incapaces de compartir una gran cantidad de este recurso.

Estos problemas y sus posibles soluciones son tratados con un poco más de profundidad en [3].

## 3.7. Cache y Memoria Virtual

### Memoria Virtual

La Memoria Virtual (o Virtual Memory) es una técnica que se utiliza para darle a los programas la ilusión de que la memoria es más grande de lo que en realidad es. La misma consiste en que los procesos utilicen un conjunto de direcciones diferentes a las del ***espacio de direcciones físicas***. Dicho conjunto recibe el nombre de ***espacio de direcciones virtuales*** [8].

Como cada una de las direcciones virtuales que estén siendo utilizadas estará asociada a una dirección física, hay que proveerles a los programas un mecanismo para ***traducir*** una dirección virtual a una dirección física cuando se acceda a la información en memoria. Dicho mecanismo consiste en revisar una tabla, la cual recibe el nombre de ***Tabla de Páginas***, donde se almacenan las distintas traducciones dirección virtual-dirección física. Dichas tablas se encuentran alojadas en memoria y son credas al mismo tiempo en que se crea el proceso que utilizará el espacio de direcciones virtuales en cuestión. A la hora de realizar las traducciones de direcciones, el encargado de llevar a cabo dicha tarea es la **MMU**. Cabe destacar que, como a cada proceso se le asigna un espacio de direcciones virtuales único, existirá una Tabla de Páginas por cada proceso que este corriendo [2].

Un problema que puede surgir al usar Memoria Virtual es que, como el espacio de direcciones virtuales puede ser mucho mayor que el espacio de direcciones físicas, habrá casos en los que todas las partes de un proceso a cargar no quepan en memoria. Dado este problema, se utiliza un espacio de almacenamiento secundario llamado ***área de intercambio*** en el cual se almacenan las partes que no sean alojadas en la memoria [1]. Cuando sea necesario utilizar una parte de un programa que este en el disco, simplemente será intercambiada por una de las partes alojadas en memoria.

**Tablas de Páginas**

Las Tablas de Páginas reciben dicho nombre dado que la Memoria Virtual es comúnmente implementada utilizando ***paginación bajo demanda*** (paginación + intercambio (swapping)) [4]. Además, el proceso de revisar una Tabla de Páginas en busca de una traducción es comúnmente llamado ***page walk***.

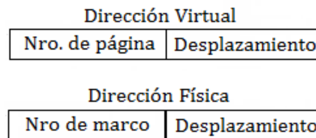


Figura 12: Forma de una dirección física y una dirección virtual al utilizar paginación.

Ahora bien, lo anterior no significa que la Memoria Virtual no pueda ser implementada en sistemas que utilicen segmentación. Por ejemplo, los micros Intel x86 trabajan con segmentación, aunque a partir de su versión 286 incluyen un modo de manejo de memoria protegido el cual les permite trabajar, de manera opcional, utilizando paginación [9]. Además, algunos sistemas presentan un esquema de segmentación paginada que consiste en dividir un segmento en páginas [4].

**Estructura de una Tabla de Páginas**

Una entrada en la Tabla de Páginas está formada por el número de página de la dirección virtual (el cual se usa como índice), un bit $P$ el cual se setea en 1 si la página del proceso asociada a la dirección virtual a traducir se encuentra alojada en la memoria (0 en caso contrario) y el número de marco de la dirección física asociada a la dirección virtual a traducir. En la figura 13 se muestra un ejemplo de una Tabla de Páginas de N entradas en la cual:

- Las direcciones virtuales 0, 1, 2, 3 y 6 están asociadas con las direcciones físicas $i$, 3, 2, 1 y 0 respectivamente.
- Las direcciones virtuales 4, 5 y $j$ están asociadas con las direcciones de disco $d_3$, $d_2$ y $d_1$ respectivamente dado que las páginas relacionadas con 4, 5 y $j$ se encuentran en disco. En este caso, también se puede optar solo por dejar seteado P en 0.
- El resto de las direcciones virtuales estarán asociadas a una dirección física o una dirección de disco dependiendo si se encuentran en alguno de los dos casos antes mencionados.

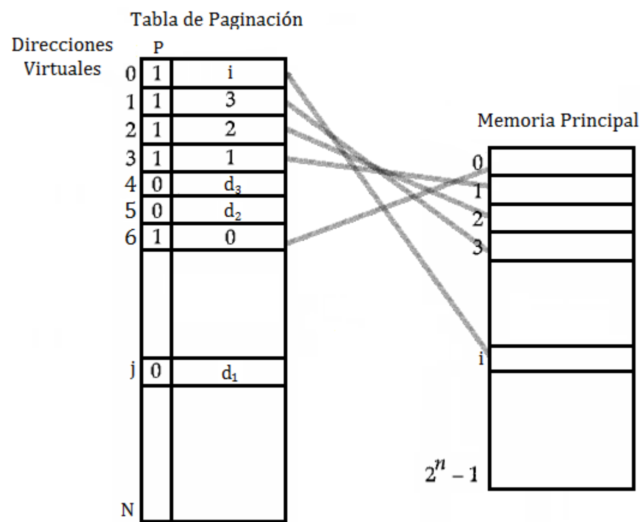


Figura 13: Tabla de Páginas

**Proceso de Traducción**

La figura 14 ilustra el proceso de traducción de una dirección virtual en una dirección física. Primero se realiza un *page walk* en busca del número de página ***vpn***. Una vez encontrado dicho número, se procede al armado de la dirección física juntando al número de marco ***fn*** con el desplazamiento D de la dirección virtual.

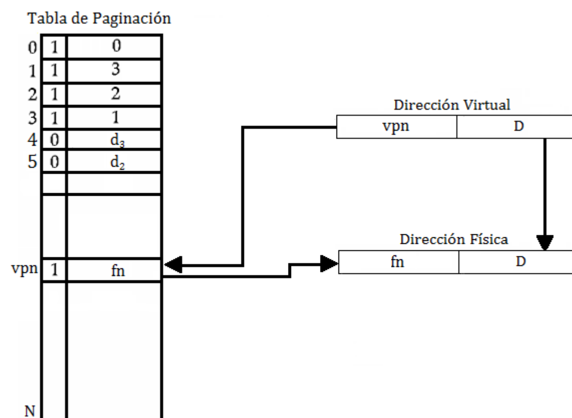


Figura 14: Traducción de una dirección virtual

**Fallo de página**

Cuando un programa haga referencia a una dirección virtual cuyo bit $P$ no este seteado en 1, la **MMU** hará que el micro levante una trap (excepción).

Dicha trap recibe el nombre de ***fallo de página*** (o ***page fault***). Luego, el sistema operativo deberá realizar una de las siguientes tareas:

- Si existe un marco libre en la memoria, traer desde el disco a dicho marco la página a la que se está haciendo referencia y después actualizar la Tabla de Páginas.
- Si la memoria está llena, escoger alguna de las páginas alojadas en la misma (ver Algoritmos de Reemplazo, sección 3.4), copiar su información a disco (solo si esta no ha sido actualizada antes de haberse producido el fallo de página), liberar la dirección física asociada a la página escogida, traer a memoria desde el disco la información relacionada a la nueva página a alojar en memoria y actualizar la Tabla de Páginas. [4]

Por otra parte, puede darse el caso en que el fallo de página sea producido por hacerse referencia a una dirección que no pertenece al espacio de direcciones virtuales. Esto significa que no puede existir una dirección física asociada a dicha dirección. Por lo tanto, el sistema operativo tendrá que terminar la ejecución del programa que haya producido dicho error de página [4].

**Problema al utilizar Memoria Virtual**

Como cada proceso podría necesitar de un espacio de direcciones virtuales muy grande, la cantidad de memoria utilizada solo para alojar a las Tablas de Páginas sería necesariamente muy grande. Esto devendría en un gran desperdicio de memoria. Para resolver el problema anterior [1] propone dos posibles soluciones:

- Como hacen casi todos los esquemas de Memoria Virtual, alojar las Tablas de Páginas en el espacio de direcciones virtuales en lugar de hacerlo en el espacio de direcciones físicas. De esto último se desprende que las Tablas de Páginas también estarán sujetas a paginación. Por lo tanto, cuando un proceso este corriendo al menos una parte de su Tabla de Páginas deberá estar alojada en memoria (incluyendo obviamente la entrada de la Tabla de Páginas para la página que este siendo utilizada por el proceso).
- Utilizar un esquema a dos niveles para organizar grandes Tablas de Páginas. En este esquema, hay un directorio de páginas en el que cada entrada señala a una Tabla de Páginas. Por lo tanto, si la longitud del directorio de páginas es X y la longitud máxima de una Tabla de Páginas es Y, un proceso puede estar formado por hasta X x Y páginas. Normalmente, la longitud máxima de una Tabla de Páginas está limitada a una página.

### Tipos de Cache

Según como sean los índices y las etiquetas de la Cache (físicos y/o virtuales) esta es llamada [2]:

- Físicamente Indexada, Físicamente Etiquetada (o **PIPT** por sus siglas en inglés)
- Virtualmente Indexada, Virtualmente Etiquetada (o **VIVT** por sus siglas en inglés)
- Virtualmente Indexada, Físicamente Etiquetada (o **VIPT** por sus siglas en inglés)

#### Cache PIPT

Este tipo de Caches utiliza tanto índices como etiquetas físicas. Esto significa que las direcciones que estas ven ya han sido traducidas por la **MMU** (i.e. son direcciones físicas). Dado lo anterior, las Caches PIPT también son llamadas Caches Físicas.

Cuando se utiliza una Cache PIPT la **MMU** se encuentra entre el micro y la antes mencionada, como lo muestra la figura 15.

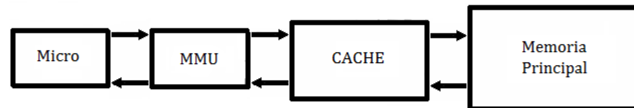


Figura 15: Cache Física

#### Cache VIVT

Este tipo de Caches utiliza tanto índices como etiquetas virtuales. Esto significa que las direcciones que estas ven aún no han sido traducidas por la **MMU** (i.e. son direcciones virtuales). Dado lo anterior, las Caches VIVT también son llamadas Caches Virtuales.

Cuando se utiliza una Cache VIVT la **MMU** se encuentra entre la antes mencionada y la memoria, como lo muestra la figura 16.

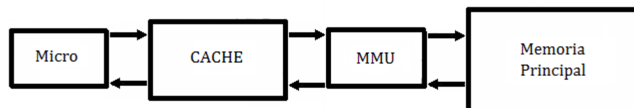


Figura 16: Cache Virtual

La traducción de una dirección virtual solo será necesaria en caso de producirse un Cache miss. Desafortunadamente, pese a que eliminar la necesidad de traducir una dirección en un Cache hit aumenta el rendimiento

de la computadora, el uso de una Cache VIVT puede complicar el diseño y/o trabajo de sistemas operativos multitareas y multiprocedadores [10]. Esto se debe a la posible aparición tanto de ***sinónimos de direcciones*** como de ***homónimos de direcciones*** en la Cache. Los primeros consisten en la asignación de nombres de dirección de distintos espacios de direcciones virtuales a la misma página. Los segundos consisten en la asignación del mismo nombre de dirección para direcciones pertenecientes a espacios de direcciones virtuales diferentes. Dicho problema puede ser resuelto simplemente vaciando la Cache cada vez que se realice un cambio de contexto. Otra forma de resolverlo es agregando un campo **ASID** (Address Space ID - Identificador de espacio de direcciones) a las direcciones virtuales [6]. De esta manera, cuando se produzca un cambio de contexto solo se quitarán de la Cache los bloques cuyas direcciones virtuales no posean el mismo campo ASID que el de la dirección virtual a la que se este haciendo referencia. Esto último se lo conoce con el nombre de ***Selective Flush*** (Vaciamiento Selectivo).

**Cache VIPT**

Las Caches VIPT buscan conseguir las ventajas de performance que ofrecen las Caches indexadas virtualmente junto con las ventajas que ofrece la simpleza de la arquitectura de las Caches etiquetadas físicamente. Por ejemplo, este tipo de Caches no tienen el problema de la posible aparición de homónimos de direcciones en ellas dado que cada dirección física tiene un *TAG* único en la Cache [2]. Como estas Caches están indexadas virtualmente y tienen etiquetas físicas, la búsqueda de un bloque en la Cache y la traducción de la dirección virtual pueden realizarse de manera simultánea. Esto es así dado que no habrá que esperar a que la traducción de la dirección haya sido realizada para conocer el índice a utilizar en la búsqueda a realizar en la Cache. Luego, deben compararse el campo *TAG* obtenido al traducir la dirección virtual y la etiqueta del bloque obtenido en la búsqueda antes mencionada.

# 4. TLB

## 4.1. Motivación

Como la traducción de una dirección virtual a una dirección de la memoria suele ser un proceso un poco costoso el cual se realiza cada vez que un proceso necesite acceder a memoria y que es muy probable que un programa acceda a la misma dirección varias veces, el mecanismo de traducción de direcciones viene acompañado de un buffer, el cual recibe el nombre de **Buffer de Traducción Adelantada**, o simplemente **TLB** por sus siglas en inglés (*Translation Lookaside Buffer*).

## 4.2. Conceptos Básicos

En la TLB se almacenan las traducciones generadas recientemente por la **MMU** para evitar que las mismas sean realizadas en reiteradas ocasiones. En otras palabras, la TLB actúa como una cache de las Tabla de Páginas.

Entrada de la TLB

| Nro. de página | Nro de marco |
|---|---|

Figura 17: Forma de una entrada de la TLB

Ahora, como lo muestra la figura 18, antes de pedirle la traducción de una dirección virtual a la **MMU** primero se chequeará si la misma se halla en la TLB, sea cual sea el tipo de Cache que se utilice.

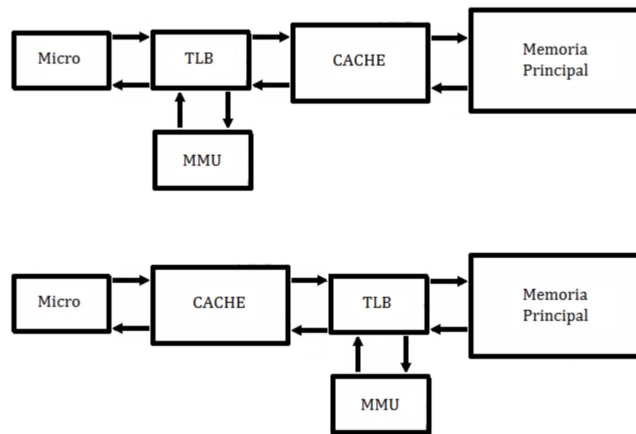


Figura 18: TLB

Si la traducción de la dirección virtual se halla en la TLB, lo que generalmente se llama ***TLB hit***, no solo no se deberá realizar dicha traducción sino que también el acceso a la misma será rápido ya que la **MMU** tomará la información directamente de la TLB. Por otro lado, cuando una traducción no se halle en la TLB, lo que generalmente se llama ***TLB miss***, la **MMU** o el sistema operativo serán los encargados de realizar dicha traducción dependiendo del tipo de TLB que se este utilizando (ver sección 4.3). Una vez realizada la nueva traducción esta será almacenada en la TLB. En caso de no haber lugar en la TLB para almacenar una nueva traducción, se emplean los mismos Algoritmos de Reemplazo que se utilizan en la Cache para proceder con dicho almacenamiento (ver sección 3.4). Además, cabe destacar que las TLB son asociativas (i.e. utilizan siempre una función de correspondencia asociativa) [1].

## 4.3. Tipos de TLB

Hay dos tipos de TLB:

- TLB manejada por hardware
- TLB manejada por software

### TLB manejada por hardware

Este tipo de TLB depende de la **MMU** para realizar las traducciones en caso de un TLB miss. Cuando se presenta a la **MMU** una dirección virtual para ser traducida, el hardware chequea si el número de página se halla presente en la TLB comparandolo con todas las entradas de manera simultánea [11]. En el caso de producirse un TLB miss, la **MMU** realizará un page walk ordinario y procederá según lo dicho en la sección 3.7 (Fallo de Página). Cabe destacar que este enfoque permite que se realicen operaciones en segundo plano (sin interrupción de tareas) [11].

### TLB manejada por software

Este tipo de TLB depende del sistema operativo para realizar las traducciones en caso de un TLB miss. Cuando se produzca un TLB miss la **MMU** en lugar de encargarse del mismo simplemente se lo pasará al sistema operativo y dejará que este último se encargue de responder. El sistema operativo accederá a la Tabla de Páginas que corresponda y realizará la traducción mediante el uso de software. Una vez hecha la traducción el sistema operativo también se encargará de las tareas de actualización de la TLB. [11]

## 4.4. Entradas inválidas en la TLB

A lo igual que las Caches VIVT, la TLB siempre está expuesta a la posibilidad de que en ella se hallen tanto sinónimos de direcciones como homónimos de direcciones. Para evitar este problema, cada vez que se produce un cambio de contexto la TLB se vacía. Otra posibilidad para evitar este problema, aunque solo[3] para las TLB manejadas por software, es la de utilizar Selective Flush (como fue explicado para las Caches VIVT en la sección 3.7). De esta manera, cuando se produzca un cambio de contexto solo quitarán de la TLB las traducciones cuyas direcciones virtuales no pertenezcan al espacio de direcciones virtuales del proceso que haya comenzado a correr.

[3] Actualmente Intel y AMD trabajan en la inclusión de etiquetas en TLBs manejadas por hardware.

# 5. Memoria Principal, Cache y TLB en entornos de Virtualización

## 5.1. Virtualización de la memoria

A la hora de crear una **VM**, uno de los recursos compartidos que debe ser virtualizado (abstraído) por el **VMM** es la memoria. Esta no suele ser una tarea sencilla, especialmente cuando se utiliza Memoria Virtual. Por ejemplo, un guest OS ejecutado dentro de una **VM** espera un espacio de direcciones físicas *zero-based*[4], como es provisto por el hardware de la computadora [12]. Para darle a cada **VM** la ilusión de que su espacio de direcciones físicas es zero-based, la memoria es usualmente virtualizada agregando un nivel extra de direcciones (y de traducción de direcciones).

Utilizando la terminología empleada en [13], llamaremos ***dirección de máquina*** a una dirección de la memoria, ***dirección física*** a una dirección del nuevo nivel de direcciones agregado a las **VMs** y ***memoria física***[5] a dicho nivel (o espacio) de direcciones. Además, si el guest OS emplea memoria virtual, aparte del traductor de direcciones "físicas a direcciones de máquina (y viceversa), habrá que agregar un traductor de direcciones virtuales a direcciones "físicas" (y viceversa). En la figura 19 ilustramos lo antes mencionado. En esta, las flechas representan la traducción de una dirección virtual/"física" en una dirección de máquina (i.e. dirección física) y la traducción de una dirección virtual en una dirección "física".

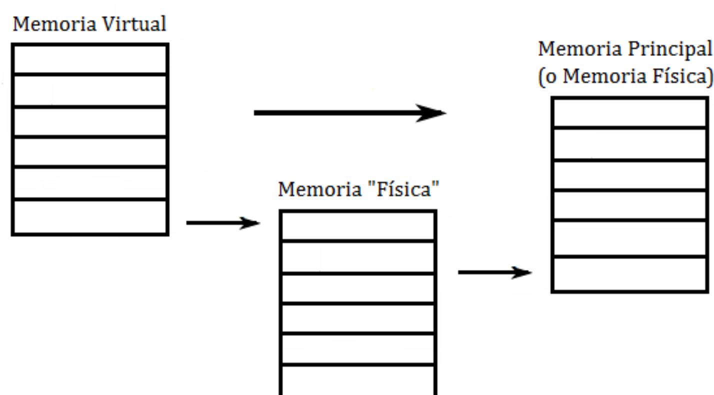


Figura 19: Memoria virtualizada

Por último, cabe destacar que a la hora de virtualizar la memoria, hay que tener en cuenta lo siguiente [14]:

[4]El elemento inicial de la memoria se encuentra en el índice cero

[5]Para evitar confusiones con la memoria real, cuando se haga referencia a la memoria física o a una dirección de dicha memoria, escribiremos memoria/dirección "física".

i) La región de memoria del **VMM** debe estar protegida (aislada) de las regiones de memoria de las de los kernels de los guest OSs y/o las aplicaciones de usuario.

ii) La región de memoria de los kernels de los guest OSs debe estar protegida de las regiones de memoria de las aplicaciones de usuario.

iii) La región de memoria de una aplicación de usuario debe estar protegida de las otras regiones de memoria de las demás aplicaciones de usuario.

## 5.2. Virtualización de la Cache

A la hora de trabajar con virtualización, la Cache no debe abstraerse. En cambio, los guest OSs la utilizan como si fuera la Cache propia de su máquina sin saber que todos ellos están utilizando en realidad la misma Cache. EL **VMM** es el encargado de que dicho uso de la Cache sea transparente para los distintos guest OSs y de que se realice de forma correcta. Ahora bien, el múltiple uso de la Cache antes mencionado podría causar problemas de performance. Dicho problema será explicado en la sección 5.6.

## 5.3. Virtualización de la TLB

De la misma manera que con la Cache, los guest OSs utilizarán la TLB como si fuera la TLB de su propia máquina, por lo que este aspecto no debe ser abstraído. Las que si deben ser abstraídas son las Tablas de Páginas y su manejo. Si la TLB es manejada por software, esto puede ser virtualizado fácilmente como se muestra en [15]. Además, si se utiliza Selective Flush para vaciar la TLB, el **VMM** y los guest OSs pueden coexistir de manera eficiente en distintos espacios de direcciones dado que no hay necesidad de vaciar la TLB en su totalidad en los cambios de contexto [16]. Por otro lado, si la TLB es manejada por hardware esta tarea no es tan sencilla dado que para lograr la mejor performance posible todas las traducciones de páginas del espacio de direcciones del proceso que se encuentre corriendo deben estar presentes en las Tablas de Páginas accesibles por el hardware [16]. Algunas plataformas (e.g. VMware) optan por utilizar ***estructuras shadow***[6] en el **VMM** y dejar que este se encargue de la comunicación con el hardware a la hora de la traducción. Otras (e.g. Xen) permiten a los guest OSs comunicarse directamente con el hardware, aunque imponen ciertas restricciones en el uso de las páginas.

[6]Las estructuras shadow son copias de estructuras que contienen datos privilegiados, los cuales no pueden ser manejados directamente por un guest OS. Estas son mantenidas por el **VMM** para simular ciertas instrucciones.

## 5.4. Ejemplos de modelados de Virtualización

En esta sección se hará mención de la manera en que algunas de las plataformas virtuales existentes modelan la virtualización de la memoria. Todas estas plataformas toman como base lo dicho en la sección 5.1. Además, también se hará mención a algunas técnicas de manejo de memoria virtualizada las cuales serán explicadas en la sección 5.5. Cabe destacar que la información sobre *Denali* está basada en [17], la información sobre *VMware* está basada en [12] y la información sobre *Xen* está basada en [16].

### Denali

Denali es una plataforma de estilo paravirtualization, desarrollada en la Universidad de Washington. Esta plataforma está basada en la virtualización de la arquitectura IA-32 (i.e. Intel x86). Su objetivo es ejecutar de manera independiente y segura aplicaciones no confiables de servidor en una sola máquina física. Esto permitiría a cualquier desarrollador agregar un nuevo servicio en la infraestructura de Internet de un tercero sin inconvenientes. Dicho objetivo es llevado acabo proveyendo ***dominios de protección ligera*** (o lightweight protection domains), i.e. contenedores rápidos para correr las **VMs**.

**Acerca de la Memoria**

Una simplificación de la Arquitectura de Memoria que realiza Denali es no proveer memoria virtual a las **VMs**. La misma está garantizada dado que Denali apunta a trabajar con aplicaciones pequeñas que no necesitan mecanismos internos de protección de memoria. Dado lo anterior, cada **VM** recibe su propio espacio de direcciones "físicas" (único). Esto produce una mejora en la performance de cada **VM** dado que no se realizarán vaciamientos de TLB al producirse cambios de contexto entre aplicaciones en la misma **VM**. Dichos espacios de direcciones contendrán dos regiones: una región para uso normal de la **VM** y otra región protegida a la que solo puede acceder el **VMM**, la cual será utilizada por el anterior como Tabla de Páginas con traducciones dirección "física"-dirección de máquina. Cabe destacar que el aislamiento entre las distintas regiones de memoria está garantizado por el uso de ***espacios de nombres privados***. Salvo la excepción de las direcciones de redes, todos los nombres provistos por el **VMM** a una **VM** son privados. Esto significa que, por ejemplo, ninguna **VM** puede construir un nombre que haga referencia a un recurso de otra **VM**.

### Xen

Xen es una plataforma de estilo paravirtualization, desarrollada en la Universidad de Cambridge. Esta tiene como objetivo principal soportar sistemas

operativos de uso cotidiano (e.g. Windows, Linux) y permitir la ejecución de aplicaciones desarrolladas para estos sistemas operativos, sin tener que modificar las mismas. Además, esta plataforma está basada en la virtualización de la arquitectura IA-32 (i.e. Intel x86).

**Acerca de la Memoria**

Xen, a diferencia de Denali, soporta el uso de memoria virtual. Los guest OSs son responsables de manejar y asignar las Tablas de Páginas (en hardware) aunque solamente tienen permiso de solo lectura sobre las páginas. En caso de querer actualizar alguna página, dicha actualización es enviada al **VMM** mediante una *hypercall*[7] para que este se encargue de la misma. Además, el **VMM** reserva una porción fija de 64 MB del tope de cada espacio de direcciones para su propio uso. Esto evita que la TLB sea vaciada cuando ocurra un cambio de contexto para entrar o salir del **VMM**. Cabe destacar que la cantidad de memoria disponible inicialmente para una **VM** es especificada durante la creación de la misma. Luego, para expandir o contraer dinámicamente dicha cantidad el **VMM** utiliza la técnica de ***Ballooning***.

### VMware

VMware es una plataforma de estilo fullvirtualization, la cual fue creada en el año 1998 por Diane Greene, Mendel Rosenblum, Scott Devine, Edward Wang y Edouard Bugnion. Esta puede encontrarse en diferentes versiones, e.g. VMware ESX Server, VMware Workstation, etc.

A continuación se analizará la versión con **VMM** *ESX Server*, i.e. VMware ESX Server. Esta versión se encuentra en uso en servidores que corren múltiples instancias de sistemas operativos (no modificados) como *Microsoft Windows 2000 Advanced Server* y *Red Hat 7.2*. Su sistema virtualiza arquitecturas IA-32 (i.e. Intel x86).

**Acerca de la Memoria**

Tres aspectos a destacar a la hora de modelar la memoria son:

i) El **VMM** mantiene una estructura de datos llamada ***pmap*** para cada una de las **VMs**. Esta estructura es utilizada para traducir direcciones "físicas" a direcciones de máquina. A su vez, ***Tablas de Páginas Shadow***, las cuales contienen traducciones de direcciones virtuales a direcciones de máquina, son mantenidas para uso del micro. Dichas tablas son conservadas de manera consistente con las traducciones en la estructura pmap. A la hora de hacer una traducción, las instrucciones de las **VMs** que manipulan las Tablas de Páginas de los guest OSs o

[7] Una hypercall es para un **VMM** lo que una llamada a sistema es para un kernel.

el contenido de la TLB son interceptadas para prevenir actualizaciones de los estados actuales de las antes mencionadas en la **MMU**. El enfoque anterior permite que las instrucciones normales de memoria sean ejecutadas sin que se produzcan gastos adicionales dado que la TLB contendrá valores de traducción de direcciones virtuales a direcciones de máquina que serían leídos de las Tabla de Páginas Shadow.

II) A la hora de asignar la cantidad de memoria a utilizar por las **VMs**, el **VMM** utiliza ***Overcommitment***. Ahora bien, el uso de esta técnica de manejo de memoria hace necesario que también sea utilizado algún mecanismo para reclamar espacio de una o más **VMs**. Una de las técnicas que el **VMM** emplea para expandir o contraer dinámicamente la cantidad de memoria asignada a una ***VM*** es ***Ballooning***. Si por algún motivo no es posible para **VMM** utilizar Balloning, este utilizará paginación bajo demanda con RAND como algoritmo de reemplazo. De todas formas, se espera que el uso de paginación sea poco común.

III) A la hora de utilizar ***Memory Sharing*** el **VMM** usa la técnica de ***Compartir Páginas Basadas en el Contenido***.

## 5.5. Técnicas para el manejo de la Memoria Virtualizada

Algunas de las técnicas más comunes para el manejo de la memoria en el **VMM** son las siguientes:

- Overcommitment
  - Reclamar Páginas
    - Introducir otro nivel de Paginación.
    - Ballooning.
- Memory Sharing
  - Compartir Páginas de Manera Transparente.
  - Compartir Páginas Basadas en el Contenido.

### Overcommitment

Esta técnica consiste en asignarle a las **VMs** una cantidad de memoria mayor al tamaño de la memoria de la computadora. Por ejemplo, en una computadora con 3 GB de memoria podemos tener 4 **VMs** de 2 GB de memoria cada una.

Dado lo anterior, el **VMM** necesita una forma de elegir que páginas y de que **VM** se hará swap a disco, hecho que se conoce como ***Reclamar Páginas***. En las plataformas de estilo fullvirtualization dicha elección puede

ser difícil dado que no se puede intervenir de manera directa con las políticas de reclamar páginas de un guest OS.

Otro problema que puede aparecer es que las políticas de reclamar páginas son muy dependientes de la implementación de cada sistema operativo, inclusive entre distintas versiones de un mismo sistema.

**Reclamar Páginas**

Una manera de reclamar páginas es ***Introducir otro nivel de paginación***, como lo hacían las primeras plataformas de virtualización [12]. Esta técnica consiste en mover algunas páginas "físicas" de una **VM** a un área de swap en el disco. Desafortunadamente, dicho nivel extra de paginación requerirá de una política de reclamar páginas de meta-nivel: ahora el **VMM** no solo se deberá elegir la **VM** a la cual revocar memoria sino que también cuales páginas en particular deberán ser reclamadas [12]. Además, dado que la paginación es transparente para los guest OSs, esta forma de reclamar páginas podría resultar en un problema de ***doble paginación*** [18].

Otra manera de reclamar páginas es ***Ballooning***. Esta técnica consiste en instalar un driver (balloon) en el guest OS el cual colabora con el VMM para reclamar páginas que son consideradas como de menor valor por el guest OS. Dicho driver obtiene la información de las páginas que fueron movidas a disco por el guest OS y las comunica al VMM para que haga lo mismo en el hardware físico. Si por alguna razón no es posible aplicar balloning, el manejo de memoria se realiza mediante un mecanismo de paginación ordinario [12].

### Memory Sharing

Memory Sharing (Compartir Memoria) no hace referencia a que dos (o más) aplicaciones o dos guest OS (o más) compartan entre si su memoria, dado que esto no cumpliría con lo mencionado en la sección 5.1. Esto hace referencia, por ejemplo, a casos en los que 2 o más **VMs** corren exactamente el mismo sistema operativo, tienen las mismas aplicaciones cargadas y/o utilizan información (posiblemente registros) en común. Entonces, dado lo anterior, lo ideal sería que en lugar de guardar en la memoria múltiples copias de lo mismo, guardar una sola copia de la aplicación o información en cuestión y que todas las **VMs** al solicitar acceso a las anteriores vean dicha copia. Una técnica que se utiliza para lograr lo antes mencionado es ***Compartir Páginas de Manera Transparente***. Esta técnica fue introducida por Disco [13] y es utilizada como un método para eliminar copias redundantes de páginas (e.g. información de solo lectura) en las distintas **VMs**. Otra posible técnica a utilizar es ***Compartir Páginas Basadas en el Contenido***. Esta técnica consiste en identificar páginas que tengan el mismo contenido (copias) y hacerlas apuntar a la misma página de máquina, la cual es marcada como ***Copy On Write*** (COW). Al intentar escribir a una página compartida se provoca una excepción que genera una copia

privada. Además, para que no se vuelva una comparación de O($n^2$), a cada página se le calcula un resumen de tipo hash y este valor se usa como clave de un diccionario [12].

## 5.6. Problemas al virtualizar la memoria y la TLB

Pese a que al utilizar virtualización se obtienen muchos beneficios, esta también puede enfrentarnos a nuevos problemas. Dichos problemas pueden darse de manera general (en todas las plataformas) o de manera particular (propios de una plataforma cualquiera).

### Problemas Generales

#### Diseño de la arquitectura x86

Los procesadores de la familia Intel x86 no fueron diseñados para ser virtualizados. Por ejemplo, estos cuentan con un conjunto de 17 instrucciones extremadamente sensibles para ser corridas en un ambiente de virtualización las cuales no emiten traps [19]. Esto hace que dichas instrucciones no puedan ser virtualizadas dado que el **VMM** no puede detectarlas. Las plataformas de estilo paravirtualization lidian con dichos problemas de diseño restringiendo o eliminando el uso de las partes no virtualizables de la arquitectura x86 en los guest OSs (e.g. Denali no ofrece a las VMs memoria virtual). En cambio, las plataformas de estilo fullvirtualization tienen una tarea más complicada a la hora de lidiar con lo anterior. Dado que no pueden realizar ninguna modificación sobre los guest OS estas deben implementar una serie de algoritmos un tanto complejos para lograr virtualizar la arquitectura x86 (e.g. VMware utiliza, entre otras cosas, la técnica *Hosted Virtual Machine Architecture* [19]).

#### Thrashing

En una máquina con varias **VMs** corriendo en ella, las cuales a su vez se encuentran corriendo un buen número de aplicaciones puede ocurrir que si la máquina host posee poca memoria o si esta posee una buena cantidad de memoria pero las aplicaciones de las **VMs** requieren el uso de mucha memoria, se produzca Thrashing (o Hiperpaginación). Se llama Thrashing a la situación en la cual la cantidad de recursos utilizados al trabajar crece pero la cantidad de trabajo a realizar disminuye. Normalmente este término se utiliza para hacer referencia a cuando se alojan y desalojan, en forma sucesiva y constante, páginas de un proceso en la memoria y/o el área de intercambio. Esto causaría que la performance del sistema operativo host o de las **VMs** caiga estrepitosamente.

### Problemas Particulares

#### Denali: Paginación dentro del VMM

Como ya se mencionó en la sección 5.4, en Denali el **VMM** se encarga de realizar la paginación hacia y desde el disco. Dado lo anterior, un atacante podría crear una **VM** con el objetivo de que esta tenga que intercambiar páginas en memoria con páginas en disco constantemente. Dicho comportamiento podría generar problemas de Thrashing.

#### Xen: Ataque con Hypercalls [20]

En Xen, un escenario típico de ataque con hypercalls puede darse en los siguientes pasos:

i) El atacante compromete una de las aplicaciones de un guest Os. Esto puedo ocurrir inclusive si el guest OS inicia en un estado limpio dado que sus aplicaciones pueden comunicarse con el mundo exterior, hecho que podría exponer a estas últimas a infecciones con malware por ejemplo.

ii) El atacante puede incrementar sus privilegios utilizando ataques comunes de llamadas a sistema dada las similitudes entre las antes mencionadas y las hypercalls.

iii) Cuando el atacante logre introducirse en el kernel del guest OS, este podrá lanzar ataques al **VMM** mediante el uso de las hypercalls.

Como las hypercalls tienen muchos privilegios estos ataques pueden ser muy efectivos. Un ejemplo de un posible ataque con hypercalls es el ***Ataque a la memoria virtualizada***. Si un atacante pudiese meter manos a las hypercalls entonces este podría causar daños como darle derechos de acceso a páginas en memoria que no estén destinadas a la aplicación comprometida por el atacante. Esto le permitiría manipular los datos manejados por otros guest OSs y también poder llegar a manipular el código del **VMM**. Además, el atacante podría ocasionar modificaciones en estructuras de memoria (e.g. las Tablas de Páginas). En [20] se proponen diferentes aproximaciones para tratar de evitar posibles ataques con hypercalls.

#### VMware: Incremento del costo de operaciones

Como ya se mencionó en la sección 5.4, el **VMM** de VMware ESX Server es el responsable de controlar el acceso a las Tablas de Páginas, validar actualizaciones y propagar cambios a las Tablas de Páginas Shadow. Ahora bien, esto incrementa (y en algunos casos de gran manera) el costo de de ejecución de ciertas operaciones del guest OS, tales como crear un nuevo espacio de direcciones virtuales o requerir propagación explícita de actualizaciones de hardware a accesos o dirty bits [16].

## Referencias

[1] William Stallings, *Operating Systems, Second Edition*. Prentice Hall.

[2] David Patterson and John Hennessy, *Computer Organization and Desing, Third Edition*. Morgan Kauffman.

[3] Alan Jay Smith, *Cache Memories*. ACM Computing Surveys, volume 14, pages 473-530. 1982.

[4] Silberschatz and Galvin, *Operating Systems Concepts, Fourth Edition*. Addison Wesley.

[5] L. A. Belady, R. A. Nelson and G. S. Shedler, *An anomaly in space-time characteristics of certain programs running in a paging machine*. Communications of the ACM, 12(6). June 1969.

[6] Dominic Sweetman, *See Mips Run, First Edition*. Morgan Kauffman.

[7] Ruud van der Pas, *Memory Hierarchy in Cache-Based Systems*. 2002.

[8] Peter J. Denning, *Virtual Memory*. ACM Computing Surveys, volume 2, pages 153-189. 1970.

[9] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Volume 3A.

[10] U. Meyer et al., *Algorithms for Memory Hierarchies*. LNCS 2625, pp. 171-192. 2003.

[11] Andrew Tanenbaum, *Modern Operating System, Second Edition*. Prentice Hall.

[12] Carl A. Waldspurger, *Memory Resource Management in VMware ESX Server*. Proceedings of the 5th Symposium on Operating Systems Design and Implementation. 2002.

[13] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum, *Disco: Running Commodity Operating Systems on Scalable Multiprocessors*. ACM Transactions on Computer Systems, 15(4), November 1997.

[14] Sangwon Seo, *Research on System Virtualization using Xen Hypervisor for ARM based secure mobile phones*. Seminar "Security in Telecommunications", Berlin University of Technology, Korea Advanced Institute of Science and Technology. January 14, 2010.

[15] D. Engler, S. K. Gupta, and F. Kaashoek, *AVM: Application-level virtual memory*. In Proceedings of the 5th Workshop on Hot Topics in Operating Systems, pages 72.77, May 1995.

[16] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield, *Xen and the Art of Virtualization*. In SOSP, pages 164-177. 2003.

[17] Andrew Whitaker, Marianne Shaw and Steven D. Gribble, *Denali: Lightweight Virtual Machines for Distributed and Networked Applications*. In Proceedings of the USENIX Annual Technical Conference. 2002.

[18] Robert Goldberg and Robert Hassinger, *The double paging anomaly*.

[19] Robert Rose, *Survey of system virtualization techniques*. 2004.

[20] Cuong Hoang H. Le, *Protecting Xen hypercalls*. Master of Science Thesis, University of British Columbia (Vancouver). July, 2009.