Tema 3 – Gestión de Memoria

Generalidades sobre gestión de memoria

Memoria virtual. Organización

Memoria virtual. Gestión

Gestión de memoria en Linux

Sistemas Operativos
Alejandro J. León Salas, 2020
Lenguajes y Sistemas Informáticos

Objetivos

- Conocer los conceptos de espacio de memoria lógico y físico y mapa de memoria de un proceso.
- Conocer distintas formas de organización y gestión de memoria que utiliza el SO.
- Saber en que consiste y para que se utiliza el intercambio de procesos (Swapping).
- Entender el concepto de **memoria virtual**.
- Conocer los esquemas de **paginación**, segmentación y segmentación paginada en un sistema con memoria virtual.
- Comprender el **principio de localidad** y su relación con el comportamiento de un programa en ejecución.
- Conocer la teoría del **conjunto de trabajo** y el problema de la **hiperpaginación** en memoria virtual.
- Conocer **como gestiona Linux la memoria** de un proceso.

Bibliografía

- [Sta2005] W. Stallings, Sistemas Operativos. Aspectos Internos y Principios de Diseño (5/e), Prentice Hall, 2005.
- [Car2007] Jesús Carretero y otros, Sistemas Operativos. Una Visión Aplicada (2 ed.), McGraw-Hill, 2007
- [Lov2010] R. Love, Linux Kernel Development (3/e), Addison-Wesley Professional, 2010.
- [Mau2008] W. Mauerer, Professional Linux Kernel Architecture, Wiley, 2008.

Contenidos

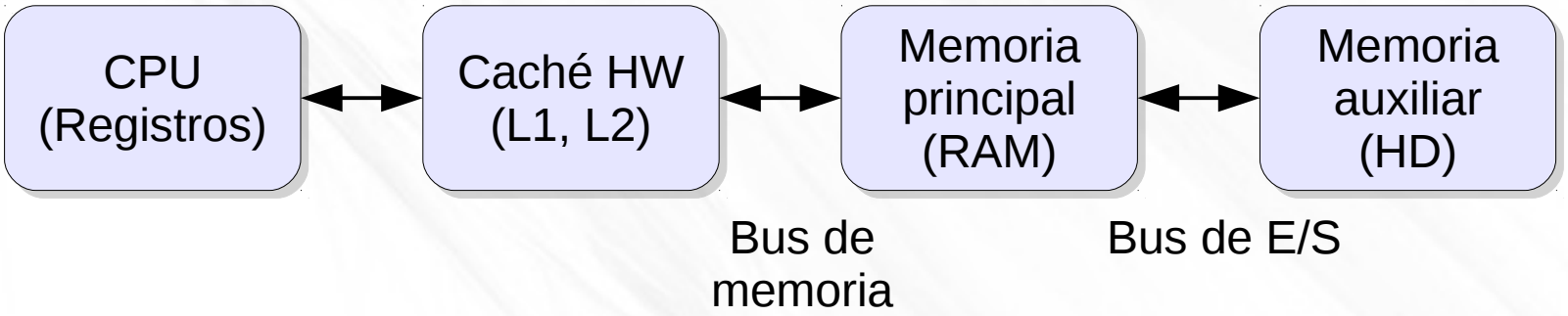
- Generalidades sobre gestión de memoria
- Organización de la Memoria Virtual
- Gestión de la Memoria Virtual
- Gestión de memoria en Linux

Generalidades sobre gestión de memoria

- Jerarquía de memoria
- Conceptos sobre cachés
- Espacios de direcciones y mapa de memoria
- Objetivos de la gestión de memoria
- Intercambio (*Swapping*)

Jerarquía de Memoria

- Dos principios generales sobre memorias:
  - Menor cantidad, acceso más rápido.
  - Mayor cantidad, menor coste por byte.
- Así, los elementos frecuentemente accedidos se ponen en memoria rápida, cara y pequeña; el resto, en memoria lenta, grande y barata.



Conceptos sobre Cachés

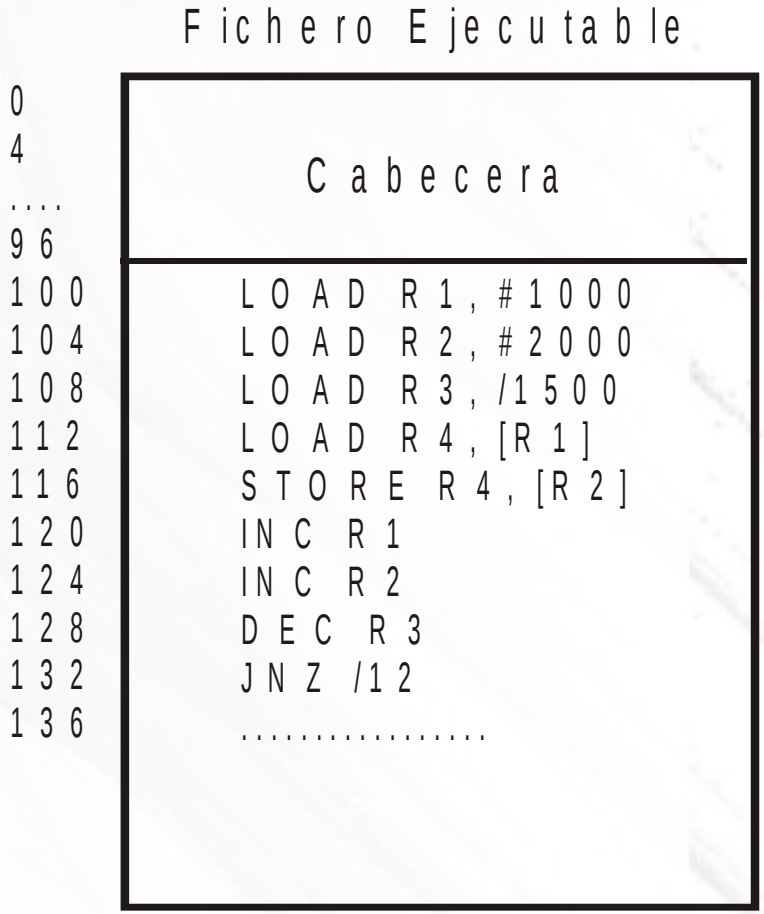
- Idea de Memoria Caché. Copia que puede ser accedida más rápidamente que el original.
- Objetivo. Hacer los casos frecuentes eficientes, los caminos infrecuentes no importan tanto.
- Acierto de caché (*cache hit*). El dato a acceder se encuentra en caché.
- Fallo de caché (*cache miss*). El dato a acceder no se encuentra en caché.

```
Tiempo_Acceso_Efectivo (TAE) = Probabilidad_acierto * coste_acierto +
Probabilidad_fallo * coste_fallo
```

- Funciona porque los programas no generan solicitudes de datos aleatorias, sino que siguen el **principio de localidad**.

Espacios de direcciones lógico y físico

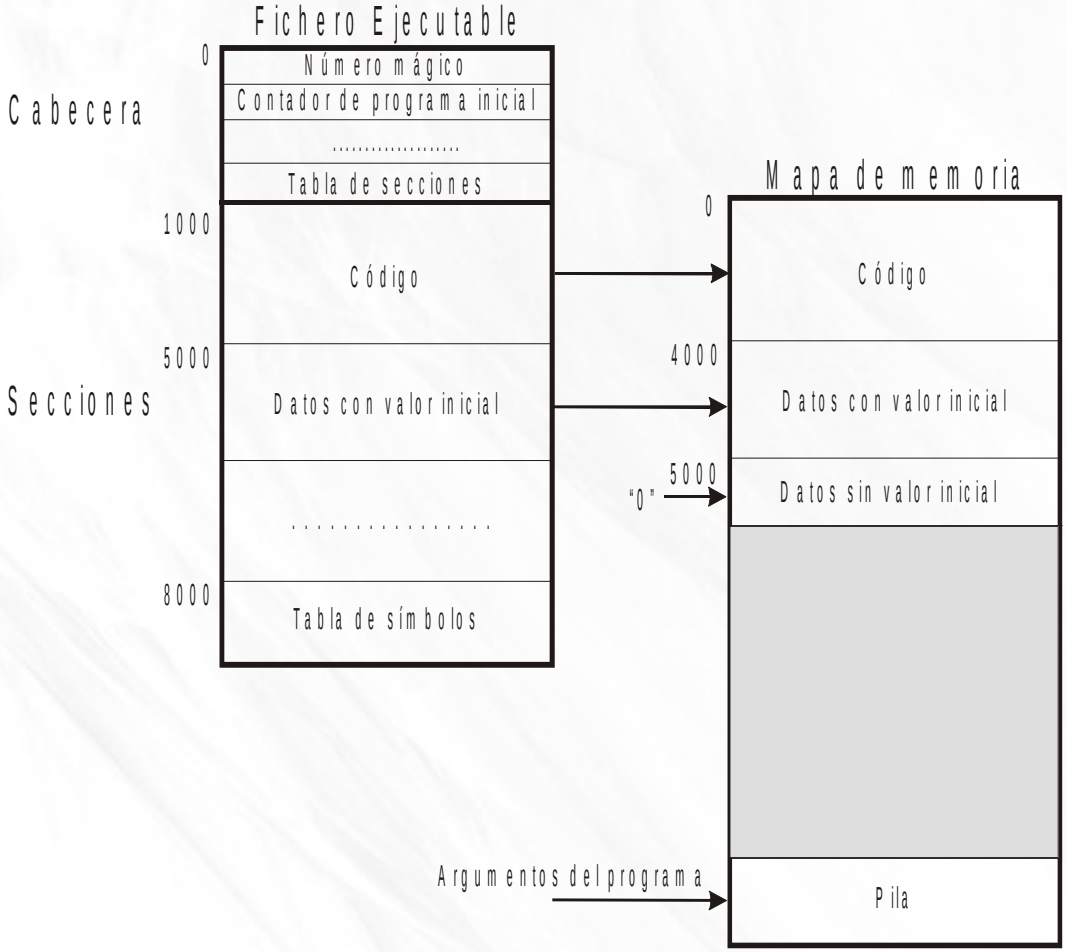
- Espacio de direcciones lógico. Conjunto de direcciones lógicas (o relativas si reubicación dinámica) (o virtuales si el sistema soporta memoria virtual) generadas por un programa.
- Espacio de direcciones físico. Conjunto de direcciones físicas correspondientes a las direcciones lógicas en un instante dado de ejecución del programa.



[Car2007], p.165

Mapa de memoria de un proceso

- Se suele definir la **imagen de un proceso** como al par formado por el mapa de memoria y el PCB.



[Car2007], p.179

Objetivos de la gestión de memoria.

- **Organización**: ¿cómo está dividida la memoria? ¿un proceso o varios procesos? Por supuesto varios.
- **Gestión**: Dado un esquema de organización, ¿qué estrategias se deben seguir para obtener un rendimiento óptimo?
  - Estrategias de asignación de memoria: Contigua, No contigua.
  - Estrategias de sustitución o reemplazo de programas (o partes) en memoria principal.
  - Estrategias de búsqueda o recuperación de programas (o partes) en memoria auxiliar.
- **Protección/compartición**
  - El SO de los procesos de usuario
  - Los procesos de usuario entre ellos

Organización.

- **Organización contigua**. La asignación de memoria para un programa se hace en un único bloque de posiciones contiguas de memoria principal (antiguo).
  - Particiones fijas
  - Particiones variables
- **Organización no contigua**. Permiten “dividir” el programa en bloques que se pueden colocar en zonas no necesariamente continuas de memoria principal.
  - Paginación
  - Segmentación
  - Segmentación paginada

¿Qué nos proporciona la paginación/segmentación sin MV?

- Todas las referencias a memoria dentro de un programa en ejecución se realizan sobre el espacio de direcciones lógico (traducción dinámica).
  - Luego, un programa puede ser retirado de memoria y al volver a traerlo puede seguir ejecutándose en una nueva área de memoria física.
- Un programa se divide en trozos (páginas o segmentos) que **NO** tienen que estar ubicados en memoria de forma contigua.
- Todos los trozos (páginas o segmentos) DEBEN residir en memoria principal durante la ejecución del programa.

Intercambio (*Swapping*).

- **Idea**. Intercambiar procesos (programas) entre memoria principal (MP) y memoria auxiliar (MA). El **proceso** pasará a estado “SUSPENDIDO_BLOQUEADO” (diseño más simple) y lo que ocupa espacio en memoria principal, el **programa**, pasará a disco.
- El almacenamiento auxiliar debe ser un disco rápido.
- El factor principal en el tiempo de intercambio es el tiempo de transferencia M. Principal ↔ M. auxiliar.
- El intercambiador (***swapper***) tiene las siguientes responsabilidades:
  - Seleccionar procesos para retirarlos de MP.
  - Seleccionar procesos para incorporarlos a MP.
  - Gestionar y asignar el espacio de intercambio.

Contenidos

- Generalidades sobre gestión de memoria
- Organización de la Memoria Virtual
- Gestión de la Memoria Virtual
- Gestión de memoria en Linux

Memoria Virtual: Organización

- Concepto de Memoria Virtual
- Unidad de gestión de memoria(MMU)
- Memoria Virtual paginada
- Tabla de páginas. Implementación
- Memoria Virtual segmentada
- Segmentación paginada

Concepto de Memoria Virtual

Concepto de Memoria Virtual (VM, *Virtual Memory*)

- Necesitamos paginación/segmentación básica.
- El tamaño del programa (código, datos, pila y resto de secciones (regiones)) puede exceder la cantidad de memoria física disponible para él.
- El número de procesos ejecutándose en MP (**grado de multiprogramación**) aumenta drásticamente.
- Resuelve el problema del crecimiento dinámico del mapa de memoria de los procesos.
- Para llevarlo a cabo se requiere usar dos niveles de la jerarquía de memoria: memoria principal y memoria auxiliar.

Concepto de Memoria Virtual

- **Idea clave**: se usan dos niveles de la jerarquía de memoria para almacenar el programa:
  - **Memoria Principal** (**MP**). Residen las partes del programa necesarias en un momento dado: **conjunto residente**.
  - **Memoria Auxiliar** (**MA**). Reside el espacio de direcciones completo del programa.
- Requisitos para su implementación:
- Disponer de la información relacionada con que partes del programa se encuentran en MP y qué partes se encuentran en MA: Tabla de Ubicación en Disco (TUD).
- Política para la resolución de un acceso a memoria situado en una parte que en ese momento no reside en MP.
- Política de movimiento de partes del espacio de direcciones entre MP y MA.

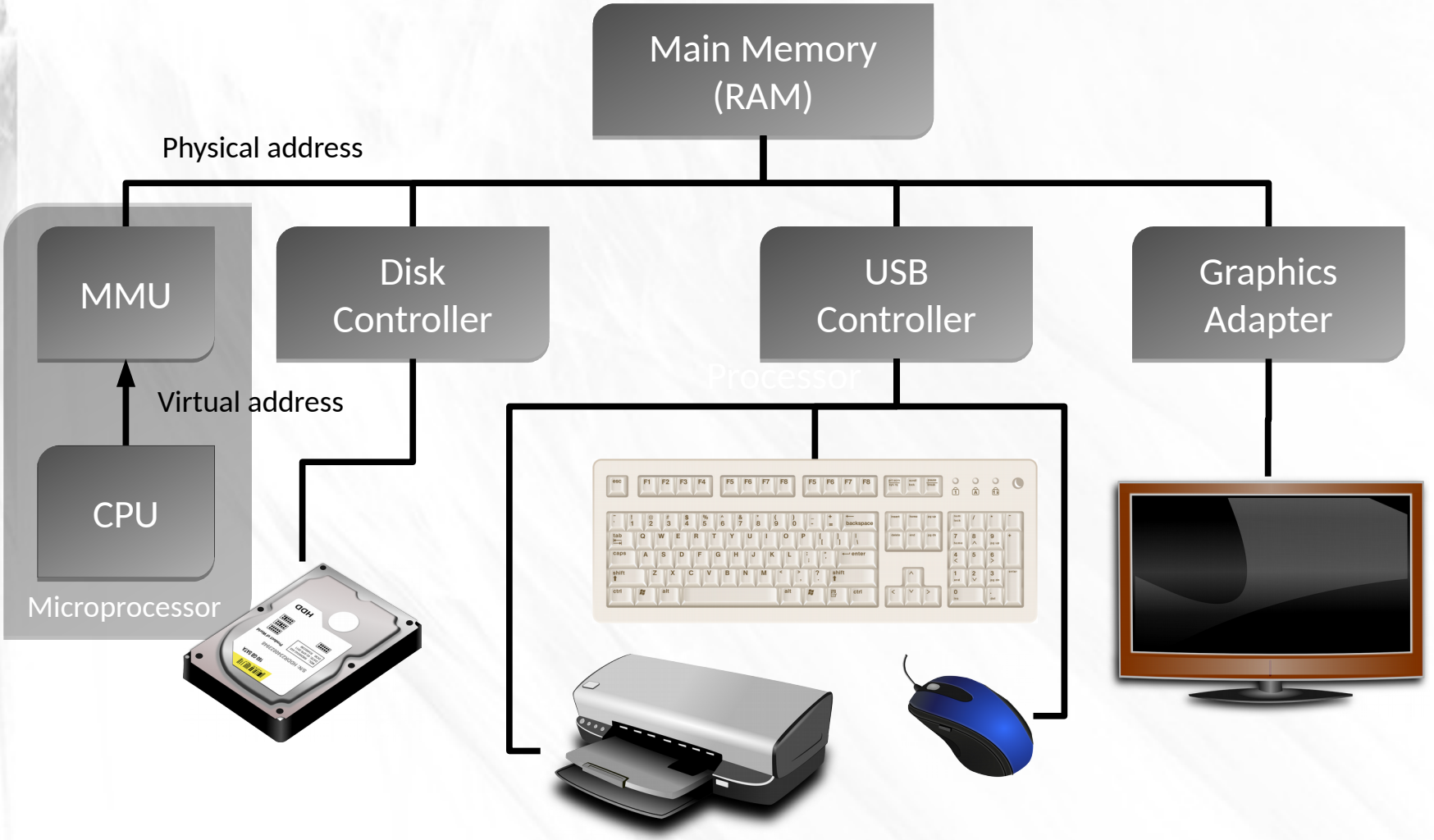
Unidad de Gestión de Memoria

- El **MMU** (*Memory Management Unit*) es un dispositivo hardware que traduce direcciones virtuales a direcciones físicas ¡Este dispositivo está gestionado por el SO!
- En el esquema MMU más simple, el valor del registro base se suma a cada dirección generada por la ejecución del programa en CPU y este resultado se utiliza en el bus de memoria para acceder a la dirección física deseada.
- El programa de usuario trata sólo con direcciones lógicas (direcciones virtuales), nunca con direcciones físicas.

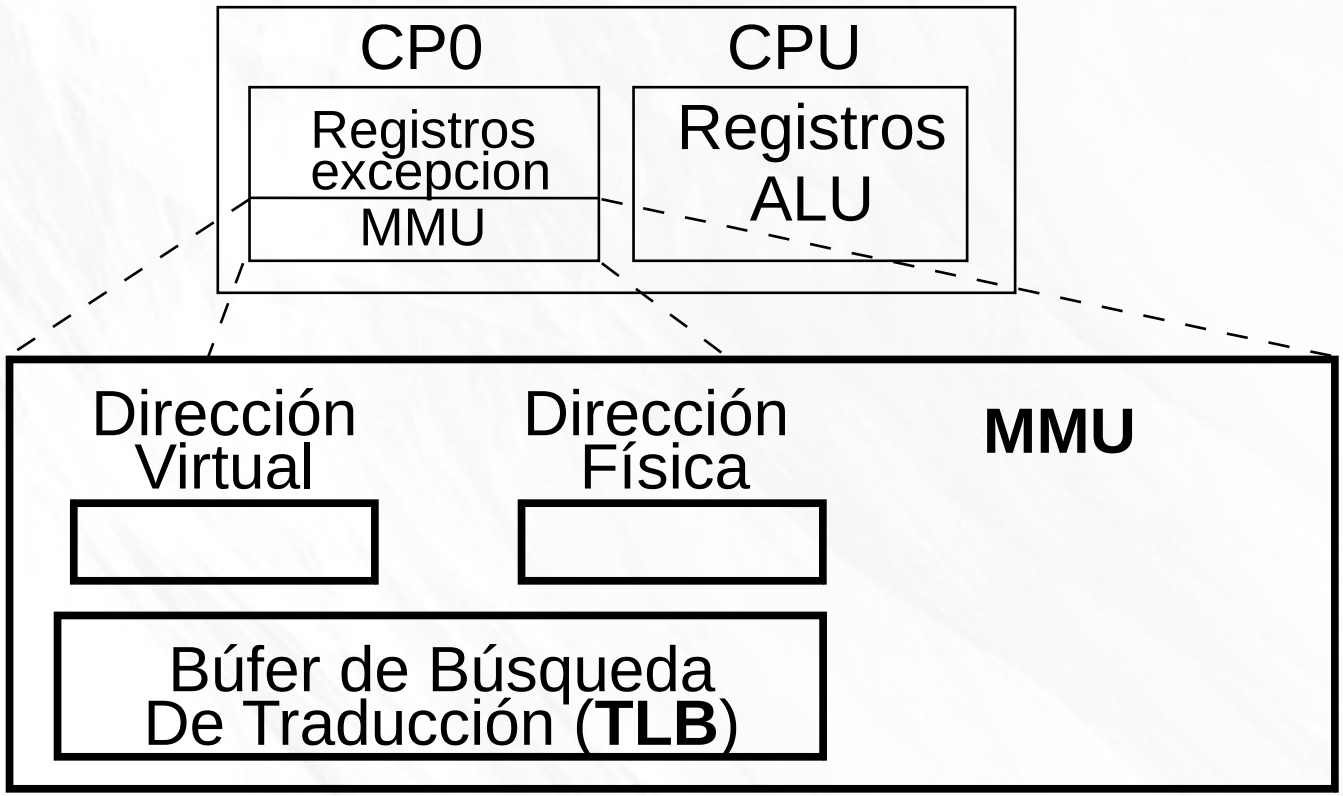
Unidad de Gestión de Memoria

- El MMU tiene unos registros TLB (*Translation Look-aside Buffer*, búfer de búsqueda de traducción previa) que mantienen la correspondencia página virtual, página física resueltas con anterioridad.
- Las responsabilidades del MMU son:
- Si acierto de TLB (*TLB hit*), realizar la traducción de dirección virtual a física.
- Si fallo de TLB (*TLB miss*), usar el mapa de memoria (Tabla de Páginas) para realizar la traducción, teniendo en cuenta que:
- Si la parte del espacio de direcciones que contiene la dirección resultado de la traducción reside en MP, cargar nuevo registro TLB y realizar traducción; si no generar EXCEPCION (*page fault exception*).

Áreas funcionales incluida la MMU



Esquema de alto nivel de MMU (MIPS R2000/3000)



Memoria Virtual Paginada

- La organización del espacio de direcciones físicas de un proceso es NO contigua.
- La memoria física se asigna mediante bloques de tamaño fijo, denominados marcos de página o páginas físicas (*physical frames*), cuyo tamaño es siempre potencia de dos (normalmente desde 0.5 a 8 KB).
- Las direcciones del espacio lógico (virtual) de un proceso se interpretan a dos niveles: los bits más significativos determinan la página virtual en la que se encuentra la dirección, y los bits menos significativos se utilizan para completar la dirección física correspondiente (*offset* en marco de página).
- La correspondencia entre “página virtual” y marco de página la almacena la entrada de la tabla de páginas: dirección base del marco de página (dirección física).

Memoria Virtual Paginada

- **Dirección virtual**. Es la que genera la CPU y **se interpreta** como un par:
  - **Número de página**, que determina la entrada de la tabla de páginas (TP) y que está representada por los bits más significativos de la dirección virtual.
  - ***Offset***, que permite completar la dirección física correspondiente a la dirección virtual, y que está representado por los bits menos significativos de la dirección virtual.
- **Dirección física**. Es la dirección real de memoria principal y se calcula en base a dos elementos:
  - Dirección base del marco de página, que almacena la "página virtual", la cual se encuentra en entrada TP.
  - *Offset*, sumado a dirección base de marco proporciona la dirección física (dirección real).

Memoria Virtual Paginada: Estructuras

- Cuando la CPU genera una dirección virtual es necesario traducirla a la dirección física correspondiente para acceder a la dirección real.
- La **Tabla de Páginas**, mantiene información necesaria para realizar dicha traducción.
- La **Tabla de Ubicación en Disco**, mantiene la ubicación de cada página en el almacenamiento auxiliar.
- La **Tabla de Marcos de Página**, mantiene información relativa a cada marco de página en el que se divide la memoria principal.

Tabla de Páginas

- La TP contiene una entrada por cada “página virtual” del proceso con los siguientes campos:
- Dirección base de marco.
- Protección, modo de acceso a la página.
- Bit de validez/presencia.
- Bit de modificación (*dirty bit*).

Nº de “página virtual”

Dirección Base de Marco de Página	Validez/ Presencia	Protección	Modificación
0x00ABC000	1	r-w	0

Tabla de Ubicación en Disco

- La TUD contiene una entrada por cada “página virtual” del proceso con los siguientes campos:
- Identificación del dispositivo lógico que soporta la memoria auxiliar. ej1. (**`Major,minor`**) de una partición de swapping. ej2. archivo de swapping.
- Identificación del bloque que contiene el respaldo de la “página virtual”.

Nº de “página virtual”

Dispositivo lógico	Nº bloque
(major=8,minor=3)	1328

Ilustración del contenido de la TP y TUD

PV	Marco	V/P	Prot.	Mod.	Dispositivo lógico	Nº bloque
0	0x20	1	r-x	0	(8,3)	1328
1	0x00	0	r-x	0	(8,3)	1329
2	0x50	1	rw-		(8,3)	1330
3	-	0	rw-		(8,3)	1331
4	-	0				
	...	0				
14	0x60	1	rw-	1		
15	0x70	1	rw-	1		

	RAM
0x00	PV1
0x10	
0x20	PV0
0x30	
0x40	
0x50	PV2
0x60	PV14
0x70	PV15
0x80	
...	

Swap device
PV0
PV1
PV2
PV3
PV14
PV15

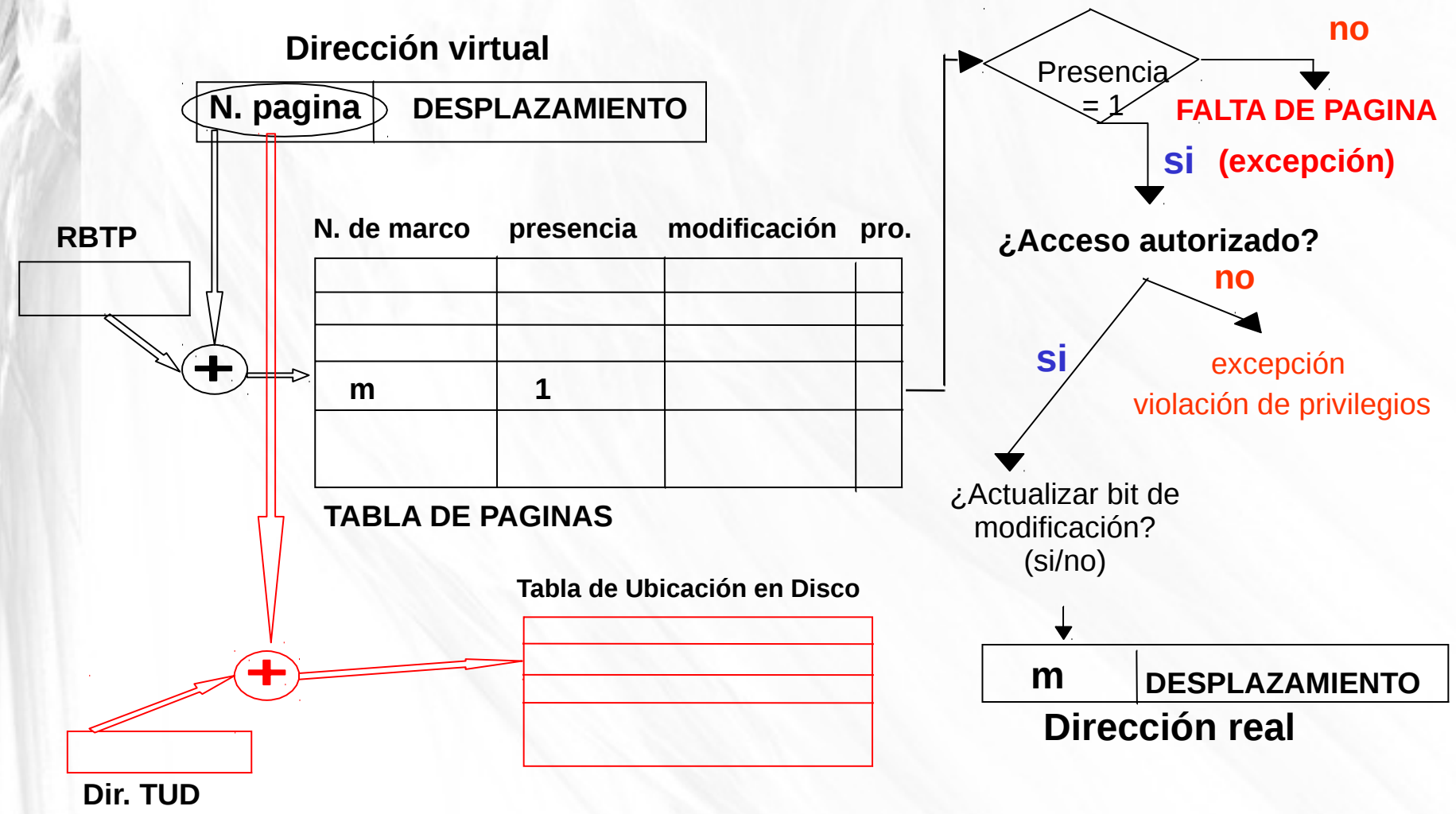
Memoria virtual mediante paginación por demanda (*demand paging*)

- La paginación por demanda es la forma más común de implementar memoria virtual.
- Se basa en el **modelo de localidad** para la ejecución de los programas:
  - Una **localidad** se define como un conjunto de páginas que se utilizan durante un periodo de tiempo.
  - Durante la ejecución de un programa, este utiliza distintas localidades.

Memoria virtual mediante paginación por demanda (*demand paging*)

- En paginación por demanda los programas residen en el dispositivo de intercambio (*backing store*) que es un HDD.
- Cuando el SO crea un proceso, antes de pasarlo a estado “LISTO”, solamente carga en memoria RAM un subconjunto de páginas del programa.
- La tabla de páginas para el proceso se inicializa con los valores correctos, páginas válidas y cargadas en RAM, páginas válidas pero no cargadas en RAM y páginas inválidas (no válidas en el espacio de direcciones del proceso).
- Tras esto el proceso ya puede cambiar a “LISTO” para poder ser elegido por el planificador de CPU (*scheduler*).

Esquema de traducción



Falta de página (*page fault*)

1. Encontrar la ubicación en disco de la página solicitada mirando la entrada de la TUD.

2. Encontrar un marco libre. Si no hubiera, se puede optar por reemplazar una página de memoria RAM.

3. Cargar la página desde disco al marco libre de memoria RAM → proceso “BLOQUEADO”.

4. FIN E/S (RSI) →

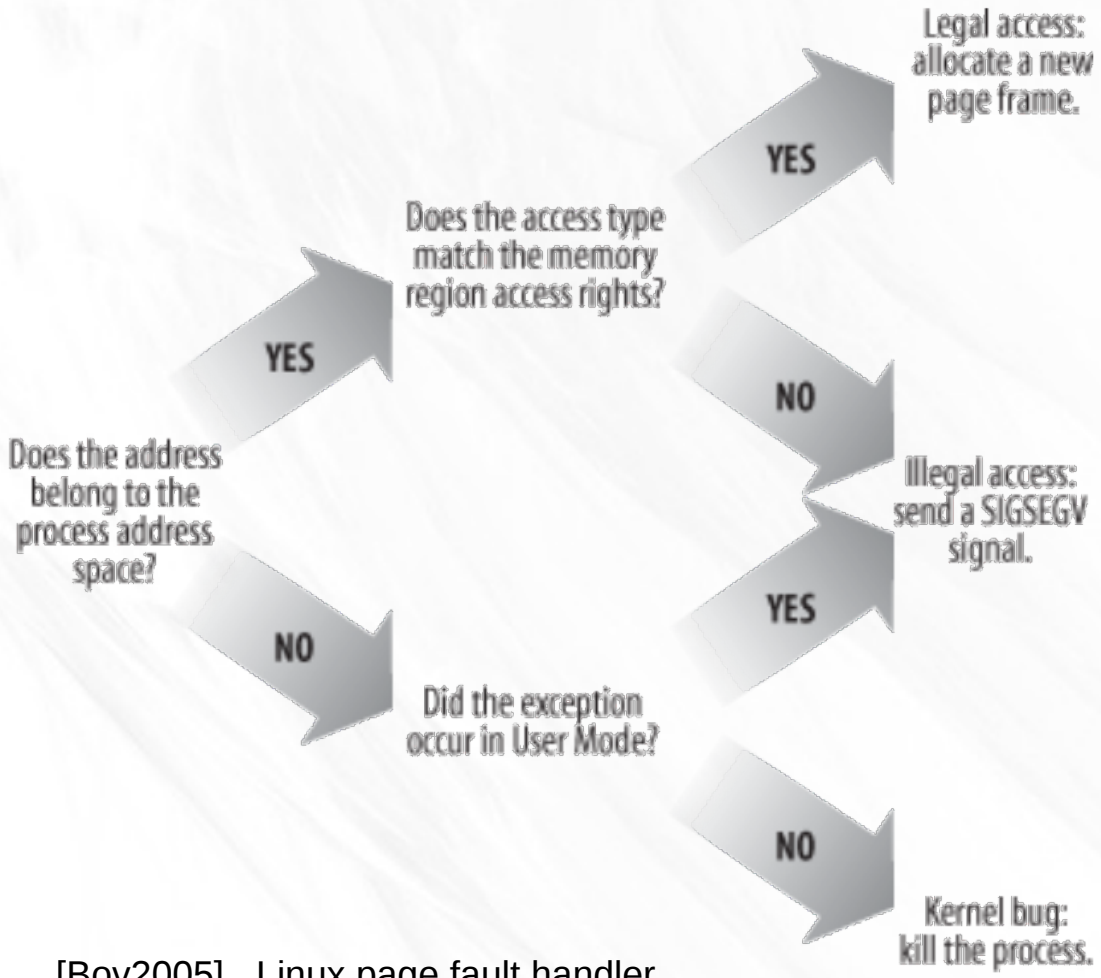
4.1. Actualizar TP(bit presencia=1, nº marco,...)

4.2. Desbloquear proceso → proceso “LISTOS”

5. Planif_CPU() selecciona proceso → Reiniciar la instrucción que originó la falta de página.

Linux. Page fault exception handler

- Distingue entre errores de programación relativos a acceso a memoria y **errores debidos a falta de página**.



[Bov2005].. Linux page fault handler.

Tabla de páginas: Implementación

- La TP se mantiene en memoria principal (*kernel*).
- El registro base de la tabla de páginas (RBTP) apunta a la TP y forma parte del contexto de registros (PCB).
- En este esquema inicial:
- Cada acceso a una instrucción o dato requiere dos accesos a memoria:
  - Acceso a la TP para calcular la dirección física.
  - Acceso a la dirección física real.
- La solución pasa por el MMU y sus registros TLB. Un acierto de TLB implica solamente un solo acceso a memoria.
- Un problema adicional viene determinado por el tamaño de la tabla de páginas.

Tamaño de la Tabla de Páginas

Ejemplo ilustrativo del problema del tamaño:

- Dirección virtual = 32 bits.
- Tamaño de página = 4 Kbytes (2^{12} bytes).
- Tamaño del desplazamiento (*offset*) = 12 bits
- Tamaño número de página virtual = 20 bits
- Nº de páginas virtuales = 2^{20} = ¡1.048.576!
- Si suponemos que el espacio ocupado por cada entrada de TP es solo el campo dirección base de marco = 32 bits = 4 bytes.
- Entonces el tamaño TP = 4.194.304 bytes = 4096 KB
- La **solución** para reducir el tamaño ocupado por la TP en memoria principal es la **Paginación multinivel**.

Paginación Multinivel

- Idea: paginar las tablas de páginas.
- Dividir la tabla de páginas en partes que coincidan con el tamaño de una página.
- Dejar partes no válidas del espacio de direcciones virtual sin paginar a nivel de página, lo que implica disponer de distintas granularidades para paginación.
- La idea es que no es necesario hacer explícita la paginación a nivel de página hasta que se habilite (haga válida) esa parte del espacio de direcciones.
- Aquellas partes del espacio de direcciones virtual que no son válidas no necesitan tener tabla de páginas.

Paginación a dos niveles

- La dirección virtual se interpreta de la siguiente forma, suponiendo que la dirección virtual tiene 32 bits, el tamaño de la página física es 4096 bytes y el tamaño de la entrada de TP ocupa 4 bytes (ejemplo anterior).
- La TP a un nivel requiere 4096 KB de espacio de memoria kernel y su dirección virtual se interpreta:

Indice en TP (PV)	offset

- La TP a dos niveles, con solo ocupación de la primera y última entrada de TP de primer nivel:
- 4096 bytes/pagina / 4 bytes/entrada = 1024 entradas.
- Si una página física contiene 1024 entradas de la TP, entonces necesitamos 2^n=1024; n=10 bits para indexar entradas y la dirección virtual se interpreta:

Indice en TP 1er nivel	Indice en Tps De 2º nivel	offset

Paginación a dos niveles

Indice en TP 1er nivel (10b)	Indice en TP 2º nivel (10b)	Offset (12b)

TP 1er nivel

0	0x00001000
1	-
2	-
3	-
...	
i	-
...	
1021	-
1022	-
1023	0x00002000

TP 2º nivel

0	0x801A1000
1	0x801B1000
2	0x801C1000
3	-
...	
j	-
...	-
1021	-
1022	-
1023	-

TP 2º nivel

0	-
1	-
2	-
3	-
...	
j	-
...	-
1021	-
1022	0x801D1000
1023	0x80000000

RAM

0x00000000	TP 1er nivel
0x00001000	TP 2º nivel
0x00002000	TP 2º nivel
...	
0x7FFFF000	
0x80000000	pila
...	
0x801A1000	texto
0x801B1000	datos
0x801C1000	datos
0x801D1000	pila
...	
0xFFFFF000	

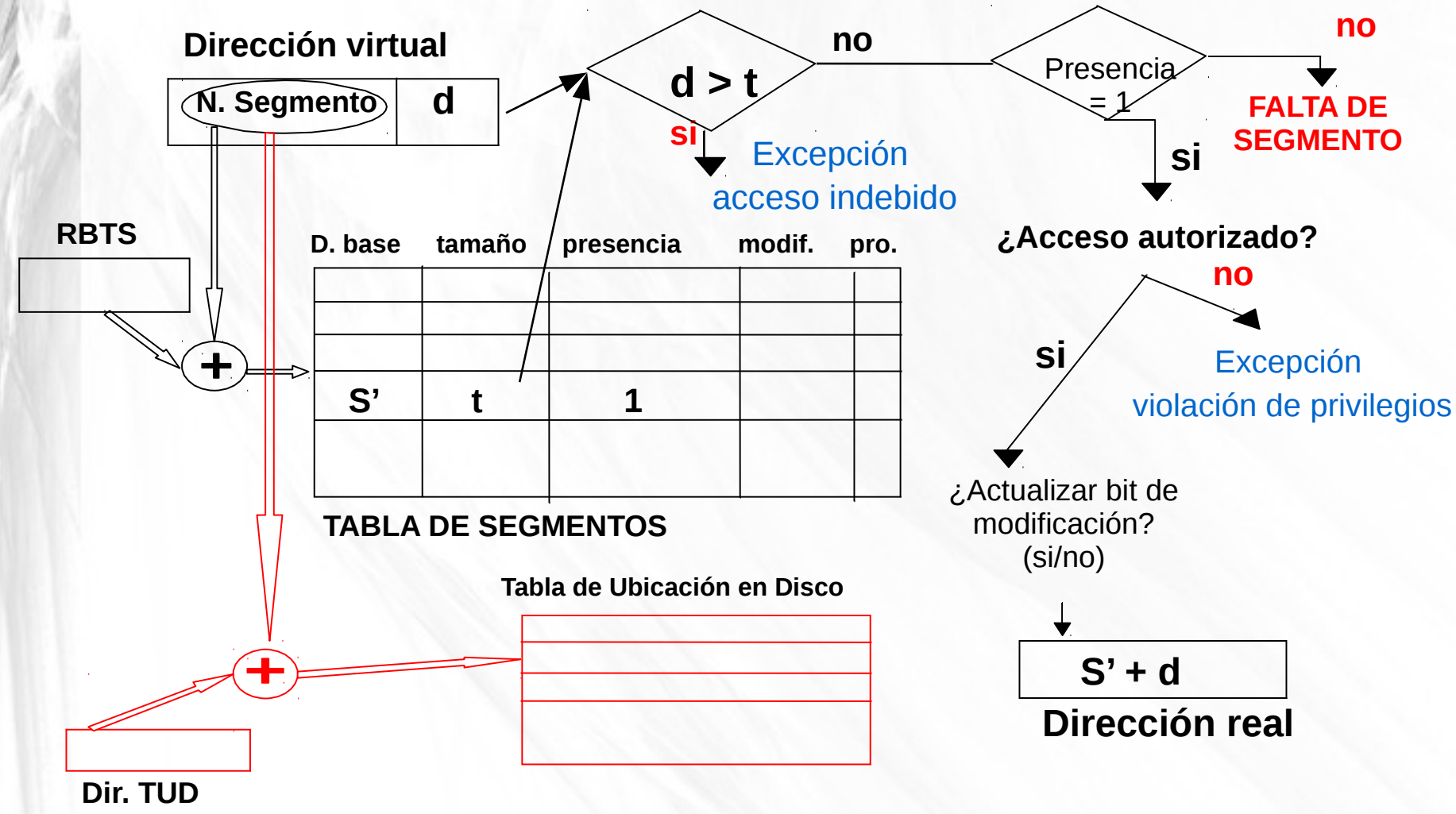
Compartición de páginas

- Una página en memoria que contenga código (texto) puede ser compartida por varios procesos.
- Las TP de los procesos que comparten una página simplemente reflejan la misma dirección base de marco.
- A la hora de seleccionar marcos de página “víctimas” para los algoritmos de reemplazo de páginas, las páginas compartidas no se seleccionan.
- Por ejemplo, el código de la **`libc`** reside en memoria principal en páginas compartidas por todos los procesos, asi como el código del programa cargador/enlazador **`ld.so`**.

Memoria Virtual Segmentada

- **Dirección virtual**. Es una tupla formada por dos elementos (id_segmento,offset), que genera la CPU y se mantiene en **dos registros distintos**:
  - **Registro de segmento**, que contiene el identificador de segmento de memoria virtual que está siendo utilizado en este momento.
  - **Registro de *offset***, que es el desplazamiento en el segmento actual e identifica la dirección virtual dentro de dicho segmento.
- **Dirección física**. Es la dirección real de memoria principal y se calcula en base a dos elementos:
  - Dirección de memoria principal en donde comienza el segmento virtual, la cual se encuentra en la entrada TS.
  - Offset, sumado a dirección física del segmento proporciona la dirección física (dirección real).

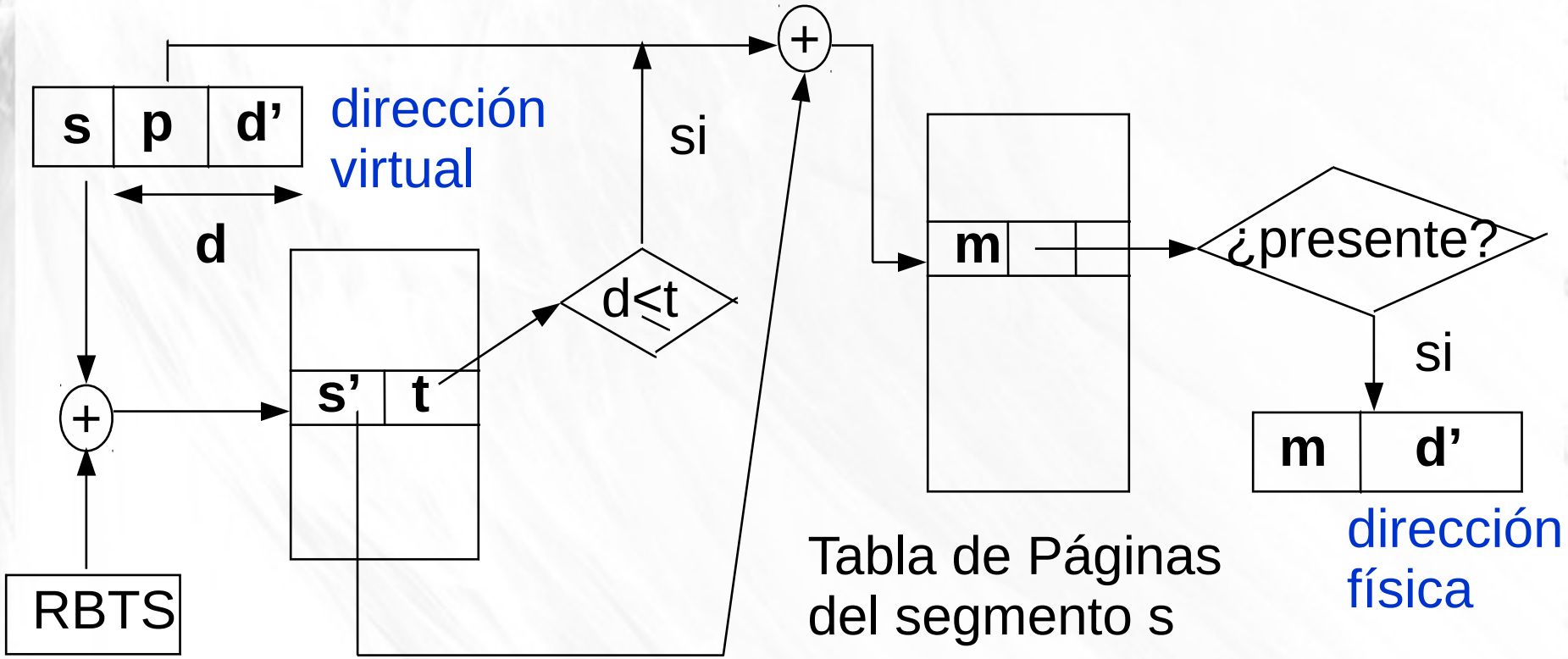
Esquema de traducción



Segmentación paginada

- La variabilidad del tamaño de los segmentos y el requisito de memoria contigua dentro de un segmento, complica la gestión de MP y MA.
- Por otro lado, la paginación simplifica la gestión pero complica más los temas de compartición y protección.
- Algunos sistemas combinan ambos enfoques, obteniendo la mayoría de las ventajas de la segmentación y eliminando los problemas de una gestión de memoria compleja.

Esquema de traducción



s' : dirección base de la Tabla de Páginas del segmento s

Contenidos

- Generalidades sobre gestión de memoria
- Organización de la Memoria Virtual
- Gestión de la Memoria Virtual
- Gestión de memoria en Linux

Memoria Virtual: Gestión

- Conceptos
- Algoritmos de sustitución
- Hiperpaginación (*thrashing*)
- Principio de localidad. Modelo del conjunto de trabajo
- Algoritmos de cálculo del conjunto residente de páginas

Conceptos

Gestión de Memoria Virtual paginada. Criterios de clasificación respecto a:

- Políticas de asignación de memoria principal: Fija o Variable
- Políticas de búsqueda (recuperación) de páginas alojadas en memoria auxiliar:
  - Paginación por demanda
  - Paginación anticipada (!= prepaginación)
- Políticas de sustitución (reemplazo) de páginas de memoria principal:
  - Sustitución local
  - Sustitución global

Conceptos

- Independientemente de la política de sustitución utilizada, existen ciertos criterios que siempre deben cumplirse:
  - Páginas “limpias” frente a “sucias”, con el objetivo de minimizar el número de transferencias MP-swap.
  - Seleccionar en último lugar páginas compartidas para reducir el número de faltas de página promedio.
  - Páginas especiales (bloqueadas). Algunos marcos de página pueden estar bloqueados por lo que no son elegibles para sustitución. Por ejemplo, búferes de E/S mientras se realiza una transferencia o búferes de cauces (**`pipe()`** o **`mkfifo()`**).

Influencia del tamaño de página

- Menor tamaño de página implica:
  - Aumento del tamaño de las tablas de páginas.
  - Aumento del nº de transferencias MP-swap.
  - Reducen la fragmentación interna.
- Mayor tamaño de página implica:
  - Grandes cantidades de información que no se están usando (¡o no será usadas!) están ocupando MP.
  - Aumenta la fragmentación interna.
- Búsqueda de un equilibrio en el tamaño de las páginas.

Algoritmos de sustitución

- Podemos tener las siguientes combinaciones en cuanto al tipo de asignación de memoria principal y tipo de sustitución de página:
  - Asignación fija y sustitución local.
  - Asignación variable y sustitución local.
  - Asignación variable y sustitución global.
- Algoritmos de sustitución:
- Óptimo. Sustituye la página que no se va a referenciar en un futuro o la que se referencie más tarde.
- FIFO. Sustituye la página más antigua.
- LRU (*Least Recently Used*). Sustituye la página que fue objeto de la referencia más antigua.
- Algoritmo del reloj.

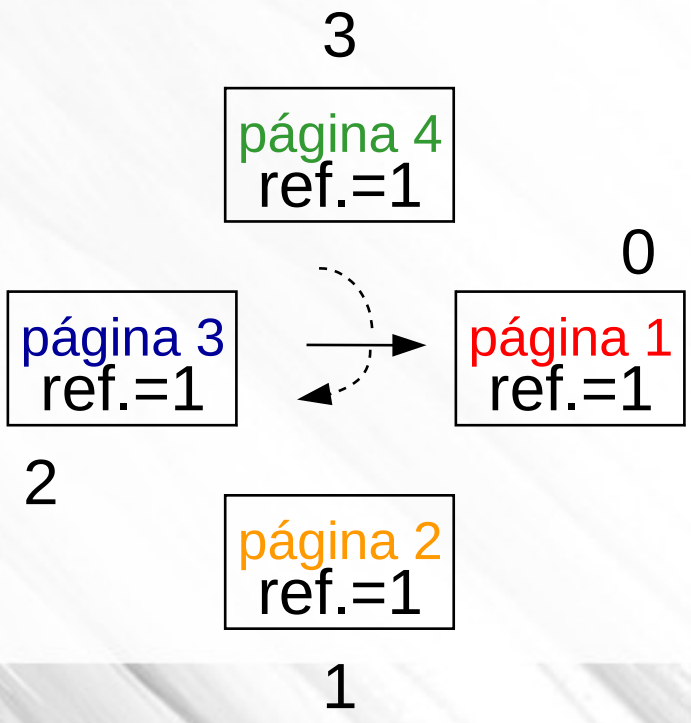
Algoritmo del reloj

- Es una aproximación al algoritmo LRU más eficiente en la que cada página tiene asociado un bit de referencia, Ref, que activa el hardware cuando se accede a una dirección incluida en la página.
- Los marcos de página se representan por una lista circular y un puntero a la página visitada hace más tiempo.
- Selección de una página:

1. Consultar marco actual

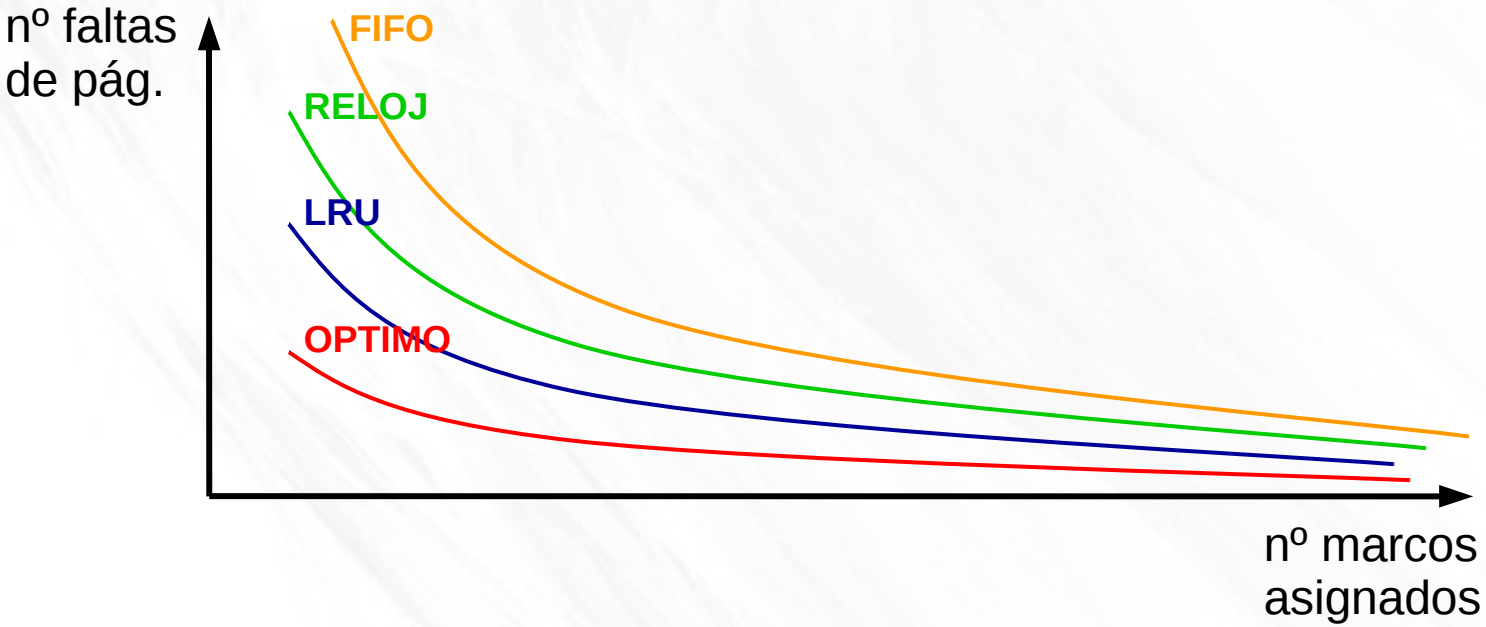
2. ¿Es R=0?

No: R=0; ir al siguiente marco y volver al paso 1.
Si: seleccionar para sustituir e incrementar posición.



Comparativa de algoritmos

- **Conclusión**. La cantidad de memoria principal disponible influye más en las faltas de página que el algoritmo de sustitución utilizado.

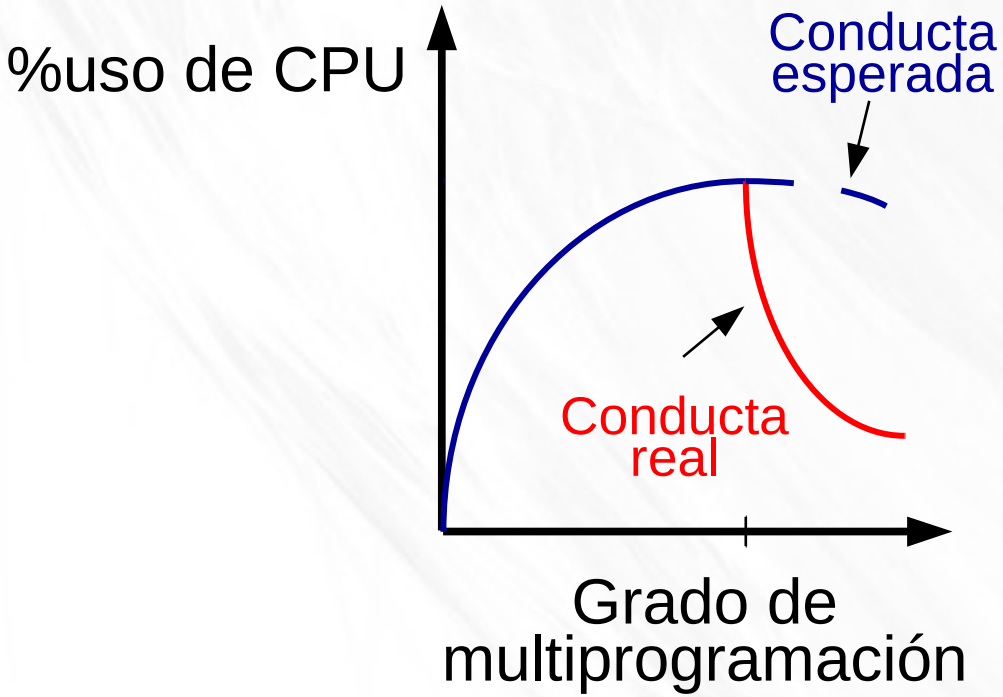


Hiperpaginación (*thrashing*)

- Un proceso está *thrashing* (hiperpaginando) si está más tiempo haciendo E/S sobre *backing store* (la tasa de faltas de página es alta) que ejecutándose.
- Si un sistema presenta suficientes procesos en este estado, se puede desencadenar el siguiente escenario:
  - Bajo uso de CPU porque falta de página → E/S.
  - Bajo uso de CPU → crea nuevos procesos para incrementar el uso de CPU.
  - Nuevos procesos que incrementan el promedio de faltas de página y entran en *thrashing*.
- Típico escenario de hiperpaginación en el que SO está resolviendo faltas de página, el tiempo de núcleo aumenta y el tiempo útil de computación cae.

Hiperpaginación (*thrashing*)

- Típico escenario de hiperpaginación en el que SO está resolviendo faltas de página, el tiempo de núcleo aumenta y el tiempo útil de computación cae drásticamente.



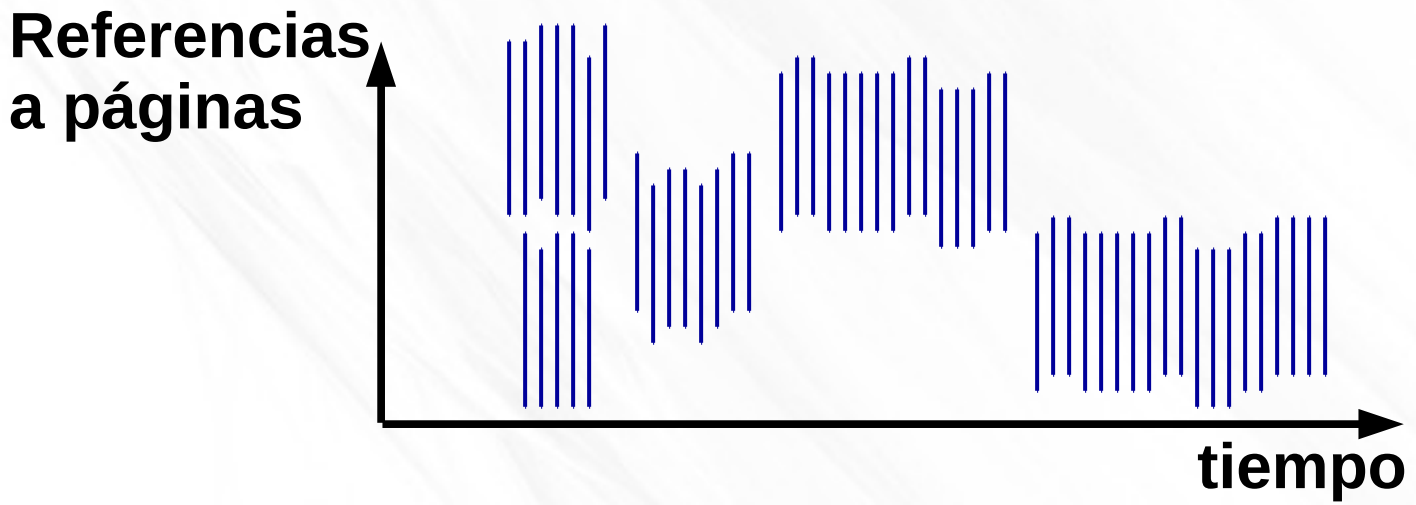
Hiperpaginación (*thrashing*)

Enfoques para evitar la hiperpaginación:

- Asegurar que cada proceso existente tenga asignado un espacio en relación a su comportamiento: Algoritmos de asignación variable de marcos, es decir de estimación del conjunto residente de páginas óptimo.
- Actuar directamente sobre el grado de multiprogramación: Algoritmos de regulación de carga.

Comportamiento de los programas

- El comportamiento de ejecución de un programa se caracteriza por la secuencia de referencias a página que realiza el proceso.
- La caracterización es importante para maximizar el rendimiento del sistema de memoria virtual: TLB, asignación, algoritmos de sustitución, etc.).



Principio de localidad

- **Principio de localidad**. Los programas referencian una pequeña parte del espacio de direcciones durante un determinado tiempo.
- Existen dos tipos de localidad: espacial y temporal.
  - **Temporal**. Una posición de memoria referenciada recientemente tiene una alta probabilidad de ser referenciada próximamente. (Ciclos, rutinas, variables globales).
  - **Espacial**. Si una posición de memoria ha sido referenciada recientemente existe una alta probabilidad de que las posiciones adyacentes sean referenciadas. (Ejecución secuencial, arrays)

Principio de localidad

- Construcciones de programación que provocan la localidad espacial y temporal:

	Espacial	Temporal
Código	Secuencia (ni bifurcación ni saltos)	ciclos
Datos	arrays	Contadores en ciclos

Modelo del conjunto de trabajo

- Definición. El conjunto de trabajo de páginas (*Working Set*), W(t,τ), de un proceso en un instante t es el conjunto de páginas referenciado por el proceso durante el intervalo de tiempo (t-τ,t).
- Mientras el conjunto de trabajo de páginas pueda residir en MP, el nº de faltas de página es pequeño.
- Si eliminamos de MP páginas del conjunto de trabajo, el número de faltas de página es alto.
- Propiedades del conjunto de trabajo:
  - Los conjuntos de trabajo son transitorios y difieren en su composición sustancialmente.
  - No se puede predecir el tamaño ni composición de un conjunto de trabajo futuro.

Modelo del conjunto de trabajo

- Requisito 1. Un proceso solamente puede ejecutarse si su conjunto de trabajo actual está en memoria principal.
- Requisito 2. Una página no puede retirarse de memoria principal si forma parte del conjunto de trabajo actual.
- El modelo representa una estrategia absoluta:
  - Si el número de páginas que referencia aumenta, y no hay espacio en memoria principal para ubicarlas, entonces el proceso se intercambia a disco.
  - La idea es que al sacar de memoria varios procesos el resto de procesos finalizan antes, incluso los que se sacaron de memoria, que en el caso de haberlos mantenido todos en memoria.

Algoritmo del conjunto de trabajo

- En cada referencia, determina el conjunto de trabajo: páginas referenciadas en el intervalo (t - τ, t] y sólo esas páginas son mantenidas en MP.

– El esquema muestra las páginas que están en MP.
– Proceso de 5 páginas
– **τ** = 4
– En t=0 WS = {A, D, E},
– A se referenció en t=0,
– D en t=-1 y
– E en t=-2

C, C, D, B, C,E, C,E,A,D

A	A	A	A	-	-	-	-	-	A	A
-	-	-	-	B	B	B	B	-	-	-
-	C	C	C	C	C	C	C	C	C	C
D	D	D	D	D	D	D	-	-	-	D
E	E	-	-	-	-	E	E	E	E	E
*		*	*	*				*	*	

Algoritmo de Frecuencia de Falta de Página (FFP)

- Idea. Para ajustar el conjunto de páginas de un proceso que residen actualmente en memoria principal (**conjunto residente**), usa los intervalos de tiempo entre dos faltas de página consecutivas:
  - Si intervalo de tiempo grande, mayor que un umbral Y, entonces todas las páginas no referenciadas en dicho intervalo son retiradas de MP.
  - En otro caso, la nueva página es simplemente incluida en el conjunto de páginas residentes.

Algoritmo de Frecuencia de Falta de Página (FFP)

- Podemos formalizar el algoritmo de la siguiente manera:

t_c = instante de tº de la actual falta de página

t_{c-1}= instante de tº de la anterior falta de página

Z = conjunto de páginas referenciadas en un intervalo de tiempo

R = conjunto de páginas residentes en MP

$$R(t_c, Y) = \begin{cases} Z(t_{c-1}, t_c) & si\ t_c - t_{c-1} > Y \\ R(t_{c-1}, Y) + Z(t_c) & en\ otro\ caso \end{cases}$$

Algoritmo de Frecuencia de Falta de Página (FFP). Ejemplo

- Garantiza que el conjunto residente crece cuando las faltas de página son frecuentes y decrece cuando no lo son.

– El esquema muestra las páginas que están en MP.
– Proceso de 5 páginas
– **Y** = 2
– La página A se referenció en t=-1
– E en t=-2
– D en t=0

D,C, C,D,B,C,E, C,E, A,D

A	A	A	A	-	-	-	-	-	A	A
-	-	-	-	B	B	B	B	B	-	-
-	C	C	C	C	C	C	C	C	C	C
D	D	D	D	D	D	D	D	D	-	D
E	E	E	E	-	-	E	E	E	E	E
*	*			*		*			*	*

Contenidos

- Generalidades sobre gestión de memoria
- Organización de la Memoria Virtual
- Gestión de la Memoria Virtual
- Gestión de memoria en Linux

Gestión de memoria en Linux

- Gestión de memoria a bajo nivel
- El espacio de direcciones de un proceso (*process address space*)
- La caché de páginas y la escritura de páginas a disco.

Gestión de memoria a bajo nivel

- El kernel gestiona el uso de la memoria física.
- Tanto el kernel como el hardware trabajan con páginas, cuyo tamaño depende de la arquitectura.
- Por ejemplo,

```
$ uname -m
x86_64
$ getconf PAGESIZE
4096
```

- Cada página física (marco de página) es representada por el kernel mediante una **`struct page`**.

Gestión de memoria a bajo nivel

- La página física es la unidad básica de gestión de memoria:
  `struct page`

```
struct page {
unsigned long flags;   // PG_dirty, PG_locked
atomic_t _count;
struct address_space *mapping;
void *virtual;
…
}
```

Gestión de memoria a bajo nivel

- Una página puede ser utilizada por:
  - La caché de páginas (*page cache*). El campo **`mapping`** apunta al objeto representado por **`struct addres_space`**(¿Qué objetos representa esta estructura? Más adelante en caché de páginas)
  - Una proyección de la tabla de páginas de un proceso.
  - El espacio de direcciones de un proceso.
  - Los datos del kernel alojados dinámicamente.
  - El código del kernel.

Gestión de memoria a bajo nivel: restricciones

- Debido a restricciones del HW, cualquier página física, debido a su dirección, no puede utilizarse para cualquier tarea.
- Por tanto, El kernel divide la memoria física en ***zonas de memoria***.
- Por ejemplo, en x86 las zonas son:

ZONE_DMA	First 16MiB of memory
ZONE_NORMAL	16MiB - 896MiB
ZONE_HIGHMEM	896 MiB - End

Gestión de memoria a bajo nivel: restricciones

- El tipo `gfp_t` permite especificar el tipo de memoria que se solicita mediante tres categorías de flags:
  - **Modificadores de acción** (`GFP_WAIT`, `GFP_IO`). Por ejemplo, `GFP_WAIT` permite que el que solicita la asignación de memoria pueda entrar en estado sleep.
  - **Modificadores de zona** (`GFP_DMA`). Por ejemplo, `ZONE_DMA` permite asignar memoria inferior a 16MB que es la única que pueden utilizar los dispositivos DMA en arquitectura x86.
  - **Tipos** (especificación más abstracta). Ejemplos de solicitud de tipos de memoria:
    - `GFP_KERNEL` indica una solicitud de memoria para kernel.
    - `GFP_USER` permite solicitar memoria para el espacio de usuario de un proceso.

Gestión de memoria a bajo nivel: API

- Interfaces para la asignación de memoria física que proporcionan memoria en múltiplos de páginas físicas.

```
struct page* alloc_pages(gfp_t gfp_mask, unsigned int order)
```

La función asigna 2^{order} páginas físicas contiguas y devuelve un puntero a la struct page de la primera página, y si falla devuelve NULL.

```
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)
```

Esta función asigna 2^{order} páginas físicas contiguas y devuelve la dirección lógica de la primera página.

Gestión de memoria a bajo nivel: API

- Interfaces para la liberación de memoria física que liberan memoria en múltiplos de páginas físicas.

```
void __free_pages(struct page* page, unsigned int order)
void free_pages(unsigned long addr, unsigned int order)
```

Las funciones liberan 2^{order} páginas a partir de la estructura página o de la página que coincide con la dirección lógica.

Gestión de memoria a bajo nivel: API

- Interfaces para la asignación/liberación de memoria física que proporcionan/liberan memoria en "*chunks*" de bytes.

```
void * kmalloc(size_t size, gfp_t flags)
void kfree(const void *ptr)
```

- Las funciones son similares a las que proporciona C en espacio de usuario: `malloc()` y `free()`.

```
#include <stdlib.h>
        void *malloc(size_t size);
        void free(void *ptr);
```

Ejemplo de código kernel

- Asignación/liberación de memoria en páginas.

```
unsigned long page;

page = __get_free_pages(GFP_KERNEL,3);

/* ‘page’ is now the address of the first of eight contiguous
pages ... */

free_pages(page,3);

/* our pages are now freed and we should no longer access the
address stored in ‘page’

*/
```

Ejemplo de código kernel

- Asignación/liberación de memoria en bytes.

```
struct example *p;
p = kmalloc(sizeof(struct task_struct),GFP_KERNEL);
if (!p)
/* handle error ... */
kfree(p);
```

Caché de bloques (*slab cache*): Organización

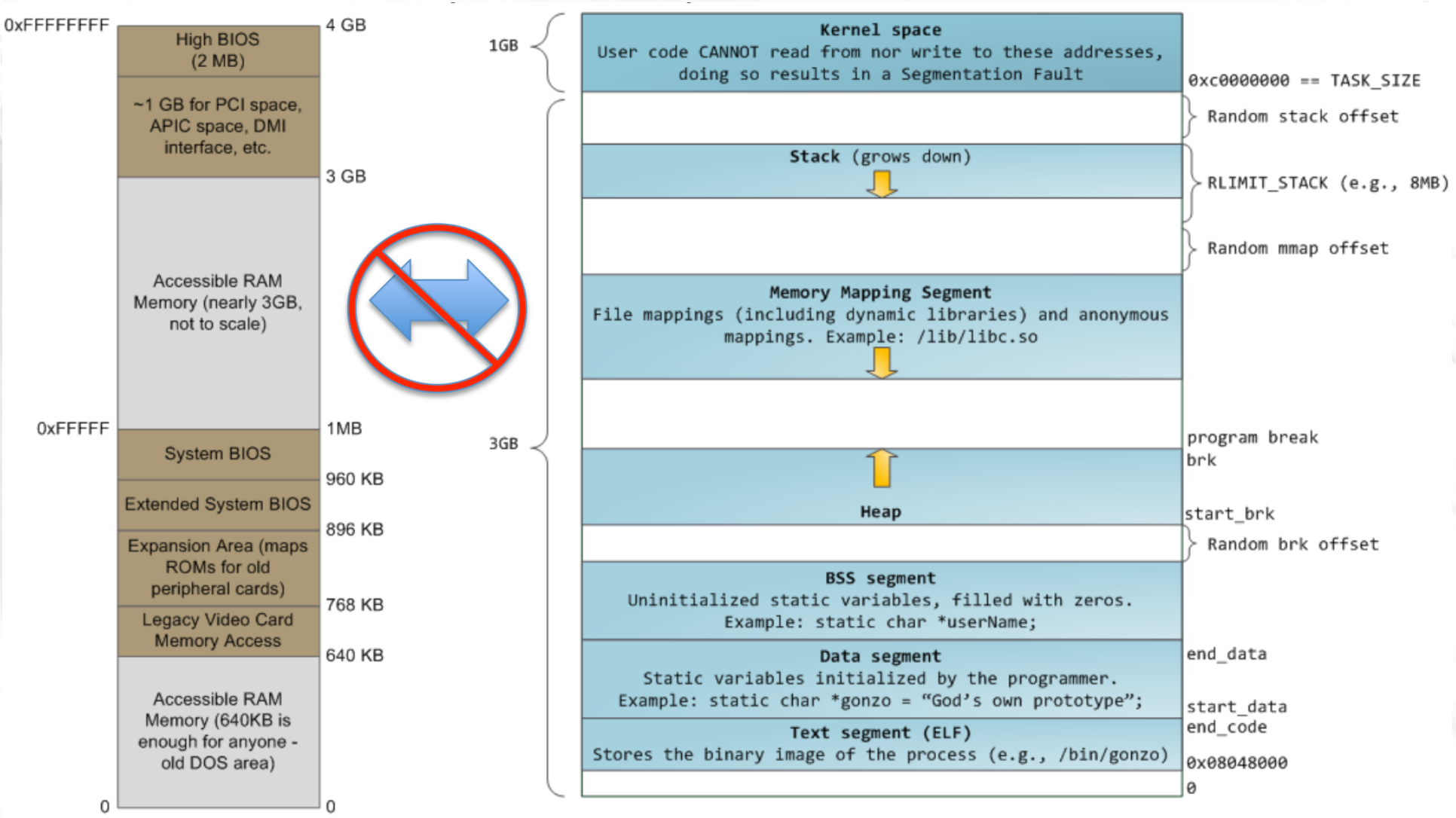
- La asignación y liberación de estructuras de datos es una de las operaciones más comunes en un kernel de SO. Para agilizar esta solicitud/liberación de memoria Linux usa el **nivel de bloques** (*slab layer*).
- El nivel de bloques actua como una caché de estructuras genérica.
  - Existe una caché para cada tipo de estructura distinta: Ejemplos, `struct task_struct` *cache,* `struct inode` *cache*.
  - Cada caché contiene multiples bloques constituidos por una o más páginas físicas contiguas ($2^{\texttt{order}}$).
  - Cada bloque (*slab*) contiene estructuras de su tipo.

Caché de bloques (*slab cache*): Funcionamiento

- Cada bloque puede estar en uno de tres estados: lleno, parcialmente lleno o vacío.
- Cuando el kernel solicita una nueva estructura:
  - La solicitud se satisface desde un bloque parcialmente lleno, si existe alguno.
  - Si no, se satisface a partir de un bloque vacío.
  - Si no existe un bloque vacío para ese tipo de estructura, se crea uno nuevo y la solicitud se satisface usando este nuevo bloque.

```
p = kmalloc(sizeof(struct task_struct),GFP_KERNEL);
```

Espacio de direcciones de proceso

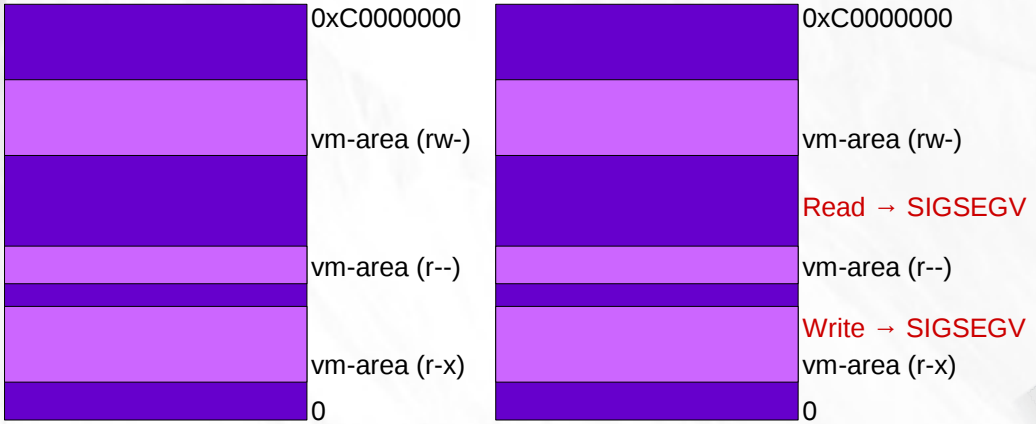


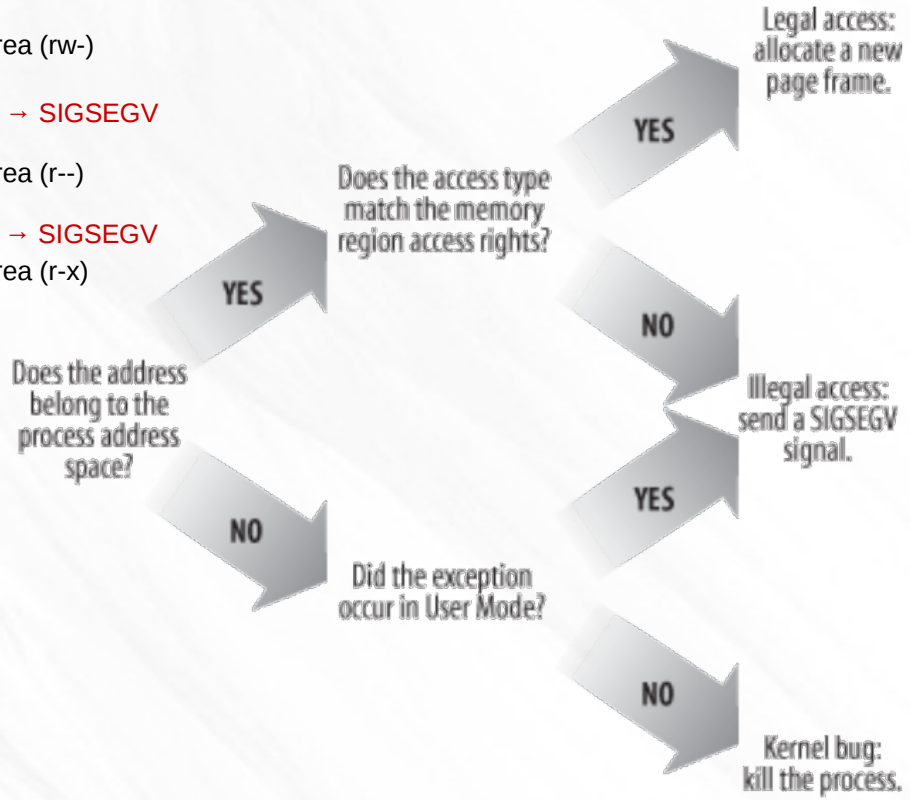
Espacio de direcciones de proceso

- En la explicación consideramos el espacio de direcciones de un proceso restringido al espacio de direcciones de los procesos ejecutándose en modo usuario. Linux utiliza memoria virtual (VM).
- A cada proceso se le asigna un espacio de memoria plano de 32 o 64 bits único. No obstante se puede compartir el espacio de memoria (CLONE_VM para hebras). Parte de este espacio solamente es accesible en **modo *kernel***.
- El proceso solo tiene permiso para acceder a determinados intervalos de direcciones de memoria, denominados **áreas de memoria (*virtual memory areas*, vm-areas)**.

Linux. Page fault exception handler

- Distingue entre **errores de programación relativos a acceso a memoria** y errores debidos a falta de página.





[Bov2005]. Linux page fault handler.

vm-area

¿Qué puede contener un área de memoria?

- Un mapa de memoria de la sección de código (*text section*).
- Un mapa de memoria de la sección de variables globales inicializadas (*data section*).
- Un mapa de memoria con una proyección de la página cero para variables globales no inicializadas (*bss section*)
- Un mapa de memoria con una proyección de la página cero para la pila de espacio de usuario.

Descriptor de memoria: struct mm_struct

- El descriptor de memoria representa en Linux el espacio de direcciones de proceso.

```
struct mm_struct {

struct vm_area_struct *mmap;  /*Lista de áreas de memoria (VMAs)*/

struct rb_root mm_rb;  /* árbol red-black de VMAs, para buscar un elemento concreto */

struct list_head mmlist;  /* Lista con todas las mm_struct: espacios de direcciones */

atomic_t mm_users; /* Número de procesos utilizando este espacio de direcciones */

atomic_t mm_count; /* Contador que se activa con la primera referencia al espacio de direcciones y se desactiva cuando mm_users vale 0 */
```

Descriptor de memoria: struct mm_struct

```
/*(cont. struct mm_struct) Límites de las secciones principales */

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 */

 /* Información relacionada con páginas */
pgd_t *pgd; /* page global directory */
unsigned long rss; /* pages allocated */
unsigned long total_vm; /* total number of pages */
}
```

Espacio de direcciones de proceso

¿Cómo se asigna un descriptor de memoria?

- Copia del descriptor de memoria al ejecutar `fork()`.
- Compartición del descriptor de memoria mediante el flag `CLONE_VM` de la llamada `clone()`.

¿Cómo se libera un descriptor de memoria?

- El núcleo decrementa el contador `mm_users` incluido en `mm_struct`. Si este contador llega a 0 se decrementa el contador de uso `mm_count`. Si este contador llega a valer 0 se libera la `mm_struct` en la caché (*slab cache*).

Espacio de direcciones de proceso

Un área de memoria (**struct vm_area_struct**) describe un intervalo contiguo del espacio de direcciones.

```
struct vm_area_struct {

struct mm_struct *vm_mm; /* struct mm_struct asociada que representa el espacio de direcciones */

unsigned long vm_start; /* VMA start, inclusive */
unsigned long vm_end; /* VMA end , exclusive */
unsigned long vm_flags; /* flags */
struct vm_operations_struct *vm_ops; /* associated ops */
struct vm_area_struct *vm_next; /* list of VMA's */
struct rb_node vm_rb; /* VMA's node in the tree */
};
```

Ejemplo de espacio de direcciones

Utilizando el archivo /proc/<pid>/maps podemos ver las VMAs de un determinado proceso.

El formato del archivo es:

start-end permission offset major:minor inode file

start-end.Dirección de comienzo y final de la VMA en el espacio de direcciones del proceso.
permission. Describe los permisos de acceso al conjunto de páginas del VMA. {r,w,x,-}{p|s}
offset. Si la VMA proyecta un archivo indica el offset en el archivo, si no vale 0.
major:minor. Se corresponden con los números major,minor del dispositivo en donde reside el archivo.
inode: Almacena el número de inodo del archivo.
file: El nombre del archivo.

Ejemplo de espacio de direcciones

```
int gVar;
int main(int argc, char *argv[])
{
while (1);
return 0;
}
```

aleon@aleon-laptop:~$ cat /proc/2263/maps

```
00400000-00401000 r-xp 00000000 08:06 5771454                    /home/aleon/vmareas
00600000-00601000 r--p 00000000 08:06 5771454                    /home/aleon/vmareas
00601000-00602000 rw-p 00001000 08:06 5771454                    /home/aleon/vmareas
7f06e683a000-7f06e69b7000 r-xp 00000000 08:05 1332123            /lib/libc-2.11.1.so
7f06e6bb6000-7f06e6bba000 r--p 0017c000 08:05 1332123            /lib/libc-2.11.1.so
7f06e6bba000-7f06e6bbb000 rw-p 00180000 08:05 1332123            /lib/libc-2.11.1.so
7f06e6bc0000-7f06e6be0000 r-xp 00000000 08:05 1332125            /lib/ld-2.11.1.so
7f06e6ddf000-7f06e6de0000 r--p 0001f000 08:05 1332125            /lib/ld-2.11.1.so
7f06e6de0000-7f06e6de1000 rw-p 00020000 08:05 1332125            /lib/ld-2.11.1.so
7fffd3dc3000-7fffd3dd8000 rw-p 00000000 00:00 0                  [stack]
```

Creación y expansión de vm-areas

¿Cómo se crea/amplía un intervalo de direcciones válido?

- `do_mmap()` permite:
  - Expandir un VMA ya existente (porque el intervalo que se añade es adyacente a uno ya existente y tiene los mismos permisos)
  - Crear una nueva VMA que represente el nuevo intervalo de direcciones

```
unsigned long do_mmap(struct file *file, unsigned long addr,
unsigned long len, unsigned long prot,
unsigned long flag, unsigned long offset)
```

Creación y expansión de vm-areas

```
unsigned long do_mmap(struct file *file, unsigned long addr,
unsigned long len, unsigned long prot,
unsigned long flag, unsigned long offset)
```

- `do_mmap()` crea una proyección de un archivo `file`, a partir del `offset` con un tamaño de `len` bytes (proyección respaldada por archivo).
- Si `file=NULL` y `offset=0` tenemos una proyección anónima.
- `addr` permite especificar la dirección inicial del espacio de direcciones a partir de la cual buscar un hueco para la nueva vm-area.
- `prot` permite especificar los permisos de acceso.
- `flag` permite especificar el resto de permisos para vm-area.

Eliminación de vm-areas

¿Cómo se elimina un intervalo de direcciones válido?

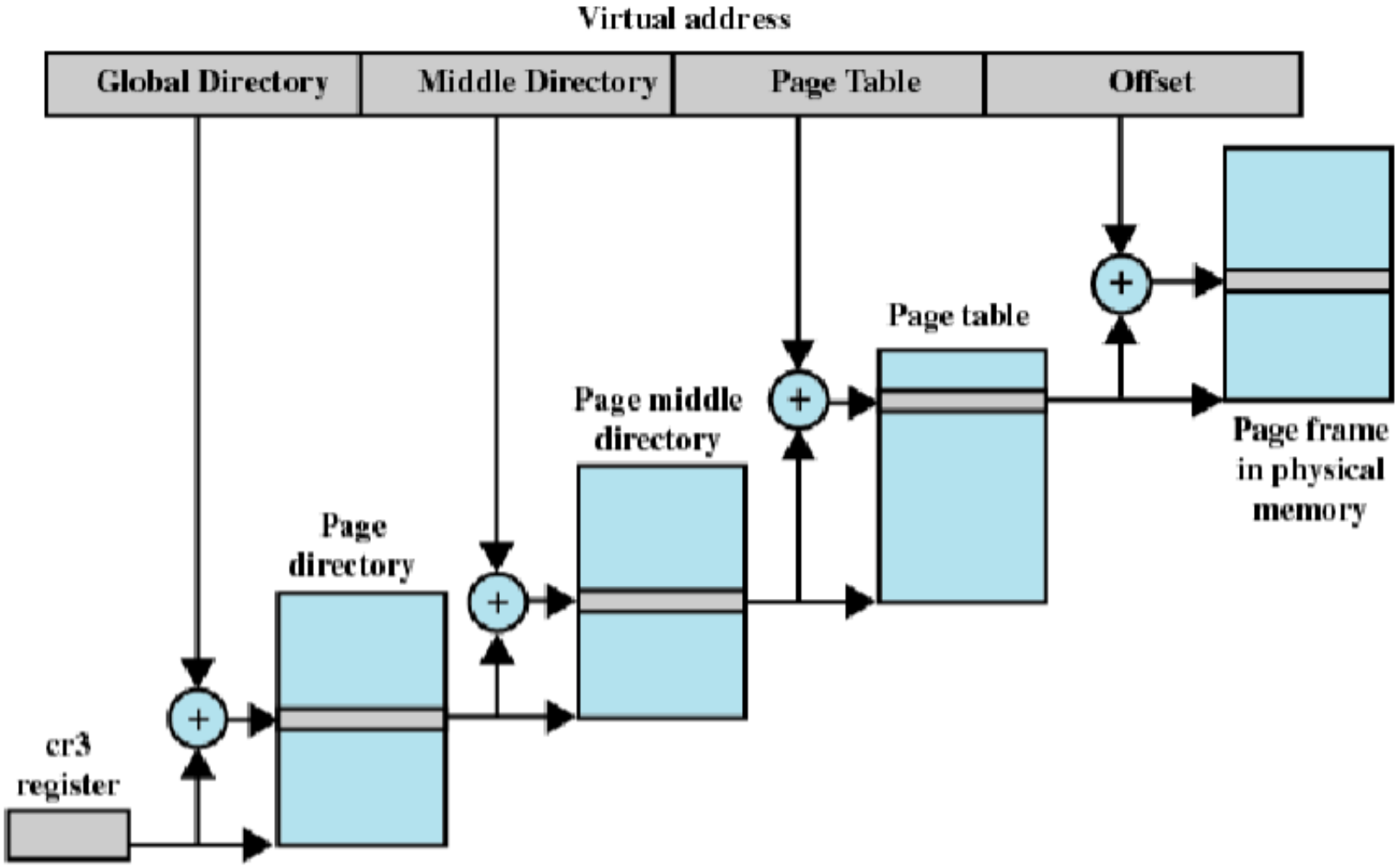
- `do_munmap()` permite eliminar un intervalo de direcciones. El parámetro `mm` especifica el descriptor de memoria (espacio de direcciones) del que se va a eliminar el intervalo de memoria que comienza en `start` y tiene una longitud de `len` bytes.

```
int do_munmap(struct mm_struct *mm,
unsigned long start,size_t len)
```

Tablas de páginas multinivel en Linux

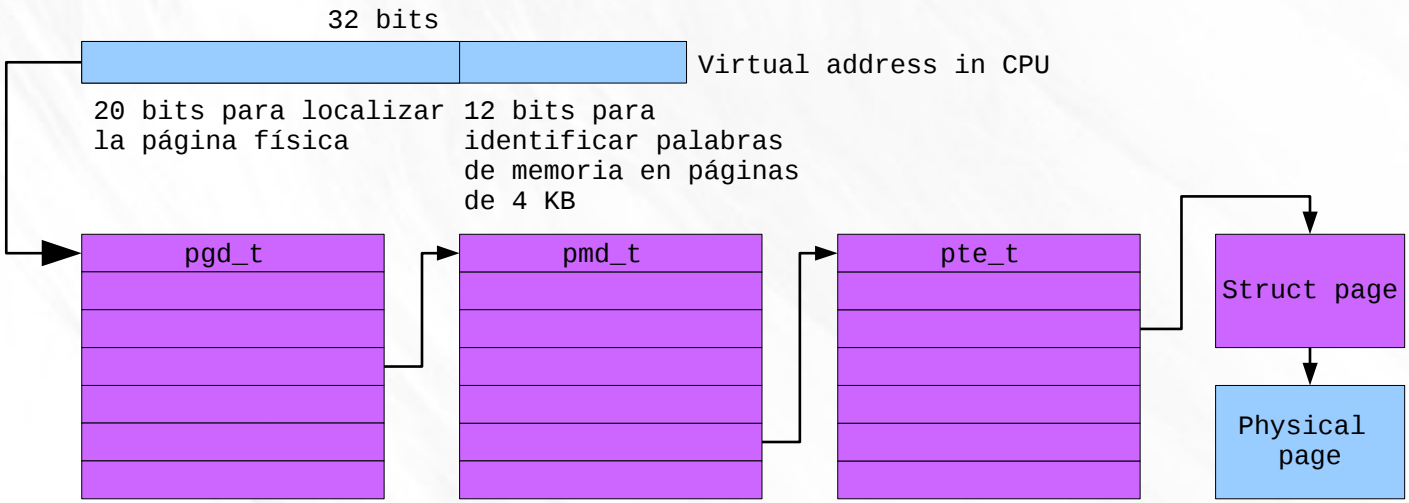
- Las direcciones virtuales deben convertirse a direcciones físicas mediante tablas de páginas. En linux tenemos 3 niveles de tablas de páginas.
  - La tabla de páginas de más alto nivel es el directorio global de páginas (del inglés page global directory, PGD), que consta de un array de tipo "pgd_t".
  - Las entradas del PGD apuntan a entradas de la tabla de páginas de segundo nivel (page middle directory, PMD), que es un array de tipo "pmd_t".
  - Las entradas del PMD apuntan a entradas en la PTE. El último nivel es la tabla de páginas y contiene entradas de tabla de páginas del tipo "pte_t" que apuntan a páginas físicas: struct_page.

Tablas de páginas multinevel de Linux



Tablas de páginas multinivel en Linux

- Pero sería mejor acceder a la palabra de memoria de una sola vez, ¿no?
- Solución, TLBs de la MMU.



Caché de páginas: Conceptos

- La caché de páginas está constituida por páginas físicas de RAM, `struct_page`, y los contenidos de éstas se corresponden con bloques físicos de disco.
- El tamaño de la caché de páginas es dinámico.
- El dispositivo sobre el que se realiza la técnica de caché se denomina almacén de respaldo (*backing store*).
- Lectura/Escritura de datos de/a disco.
- Fuentes de datos para la caché: archivos regulares, de dispositivos y archivos proyectados en memoria.

Caché de páginas: Idea

- Cuando el kernel necesita leer algo de disco primero comprueba si los datos están en la caché de páginas:
  - Si *cache hit* → leer directamente los datos de la caché.
  - Si *cache miss* → solicitar E/S de disco.
- Cuando el kernel necesita escribir algo en disco tiene dos estrategias:
  - *Write-through cache*. Actualizar memoria y disco.
  - *Write-back cache* (Linux). Las escrituras se realizan en la caché de páginas (activar flag PG_DIRTY).

Desalojo de la caché de páginas: *cache eviction*

- Proceso por el cual se eliminan datos de la caché junto con la estrategia para decidir cuáles datos eliminar.
- Linux selecciona páginas limpias (no marcadas `PG_dirty`) y las reemplaza con otro contenido.
- Si no existen suficientes páginas limpias en la caché, el kernel fuerza un proceso de escritura a disco para hacer disponibles más páginas limpias.
- Ahora queda por decidir que **páginas limpias** seleccionar para eliminar (**selección de víctima**).

Selección de víctima

- *Least Recently Used* (LRU). Requiere mantener información de cuando se accede a cada página y seleccionar las páginas con el tiempo de acceso más antiguo. El problema es cuando se accede una única vez a un archivo.
- Linux soluciona el problema usando dos listas pseudo- LRU balanceadas: *active list* e *inactive list*.
  - Las páginas de la *active list* no pueden ser seleccionadas como víctimas.
  - Solamente se añaden nuevas páginas a la *active list* si son accedidas mientras residen en la *inactive list*.
  - Las páginas de la *inactive list* pueden ser seleccionadas como víctimas.

Caché de páginas: Lectura

- La caché de páginas de Linux usa un objeto para gestionar entradas de la caché y operaciones de E/S de páginas: la `struct address_space`, que representa las páginas físicas de un archivo.

```
struct address_space {
struct inode *host; /* owning inode */ ...
};
```

- Operación de lectura implica buscar primero la información en la caché de páginas: `find_get_page()`

```
struct page* find_get_page(struct address_space,long int offset)
```

- Si devuelve `NULL` el kernel asigna una nueva página y la añade a la caché de páginas → Operación de lectura de disco

Caché de páginas: Escritura

- Dos posibilidades dependiendo del objeto que representa la `struct address_space`:
- Si representa una proyección a memoria de un archivo, se activa el flag `PG_DIRTY` de la `struct_page` y ya está.
- Si representa un archivo, entonces se busca la página en la caché de páginas, y si no se encuentra se asigna una entrada y se trae el trozo de archivo correspondiente a una página, `struct_page`.
- Se escribe la información en la página y se activa el flag `PG_DIRTY` de la `struct_page`.

Caché de páginas: *Flusher threads*

- La escritura real a disco de páginas “sucias” (`PG_DIRTY`) ocurre en tres situaciones:
  - Cuando la memoria disponible cae por debajo de un umbral de tamaño.
  - Cuando las páginas “sucias” superan un umbral de tiempo.
  - Cuando un proceso invoca las llamadas `sync()` o `fsync()`.
- Las *flusher threads* se encargan de esto:

```
If (size(free_memory) < dirty_background_ratio)
wakeup(flusher);

If (dirty_expire_interval == TRUE) wakeup(flusher);
```