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

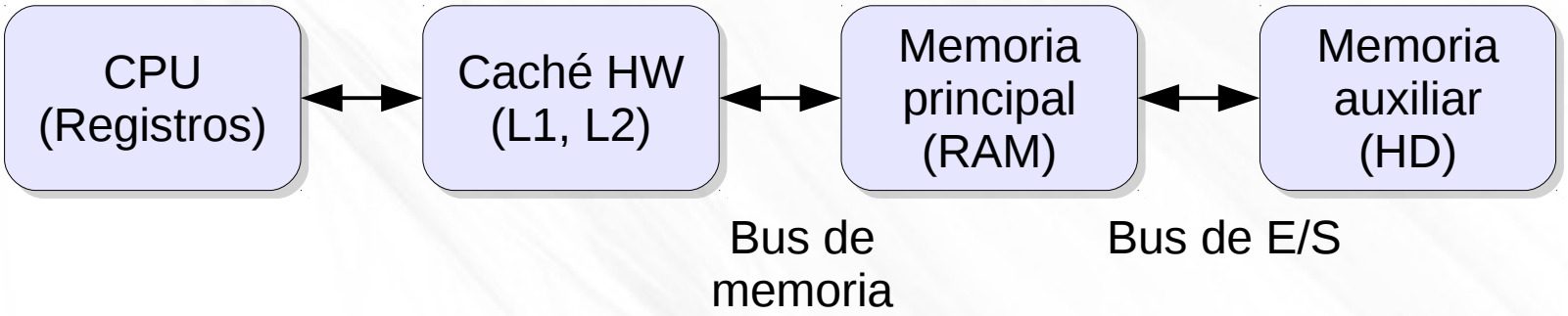
- Generalidades sobre gestión de memoria
- Organización de la Memoria Virtual
- Gestión de la Memoria Virtual
- Gestión de memoria en Linux

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

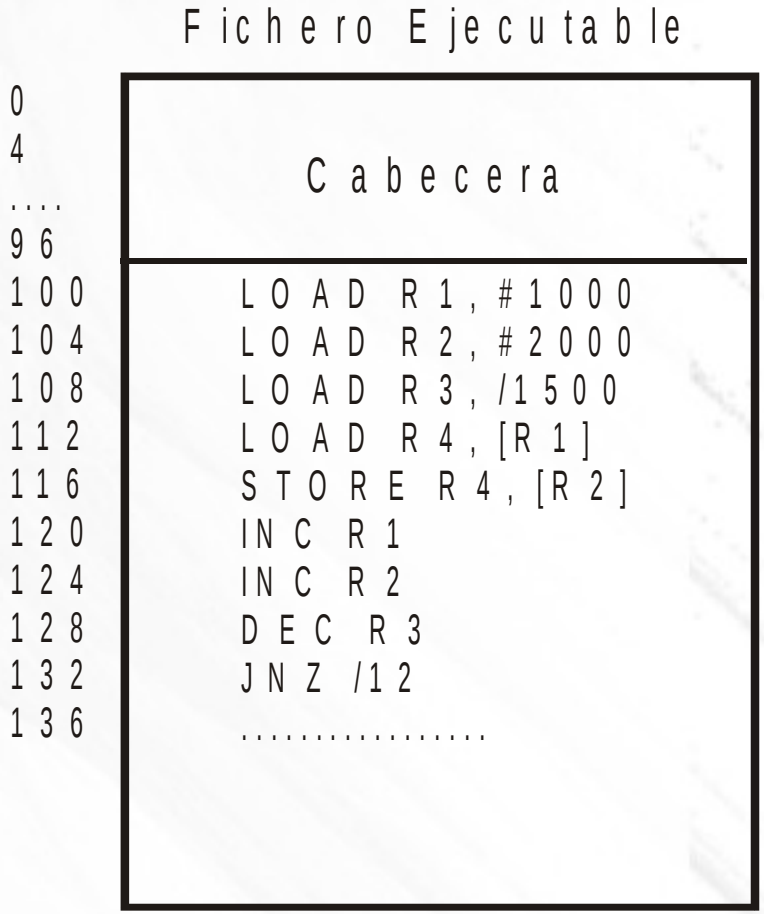
- Idea de Memoria Caché. Copia que puede ser accedida más rápidamente que el original.
- Objetivo. Hacer los casos frecuentes eficientes, los caminos infrecuentes no importan tanto.
- Acierto de caché (*cache hit*). El dato a acceder se encuentra en caché.
- Fallo de caché (*cache miss*). El dato a acceder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

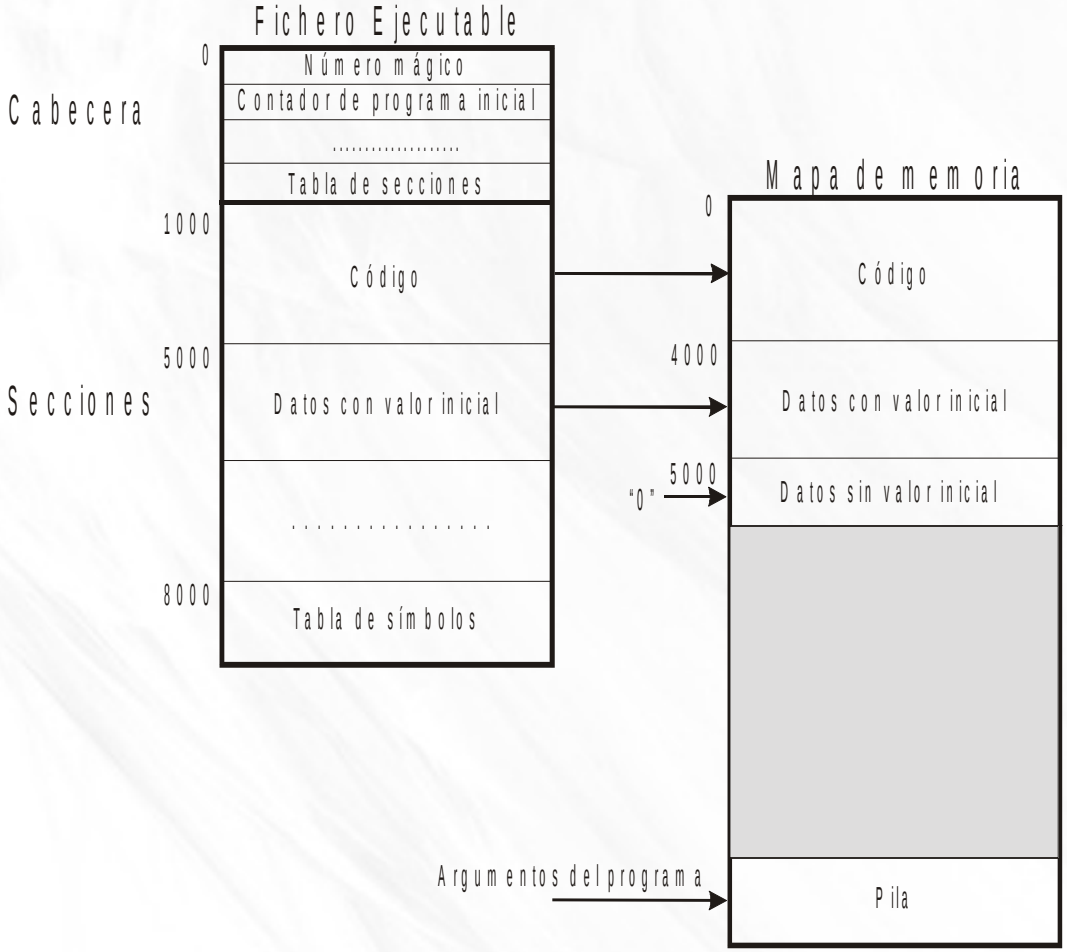
- Espacio de direcciones lógico. Conjunto de direcciones lógicas (o relativas si reubicación dinámica) (o virtuales si el sistema soporta memoria virtual) generadas por un programa.
- Espacio de direcciones físico. Conjunto de direcciones físicas correspondientes a las direcciones lógicas en un instante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

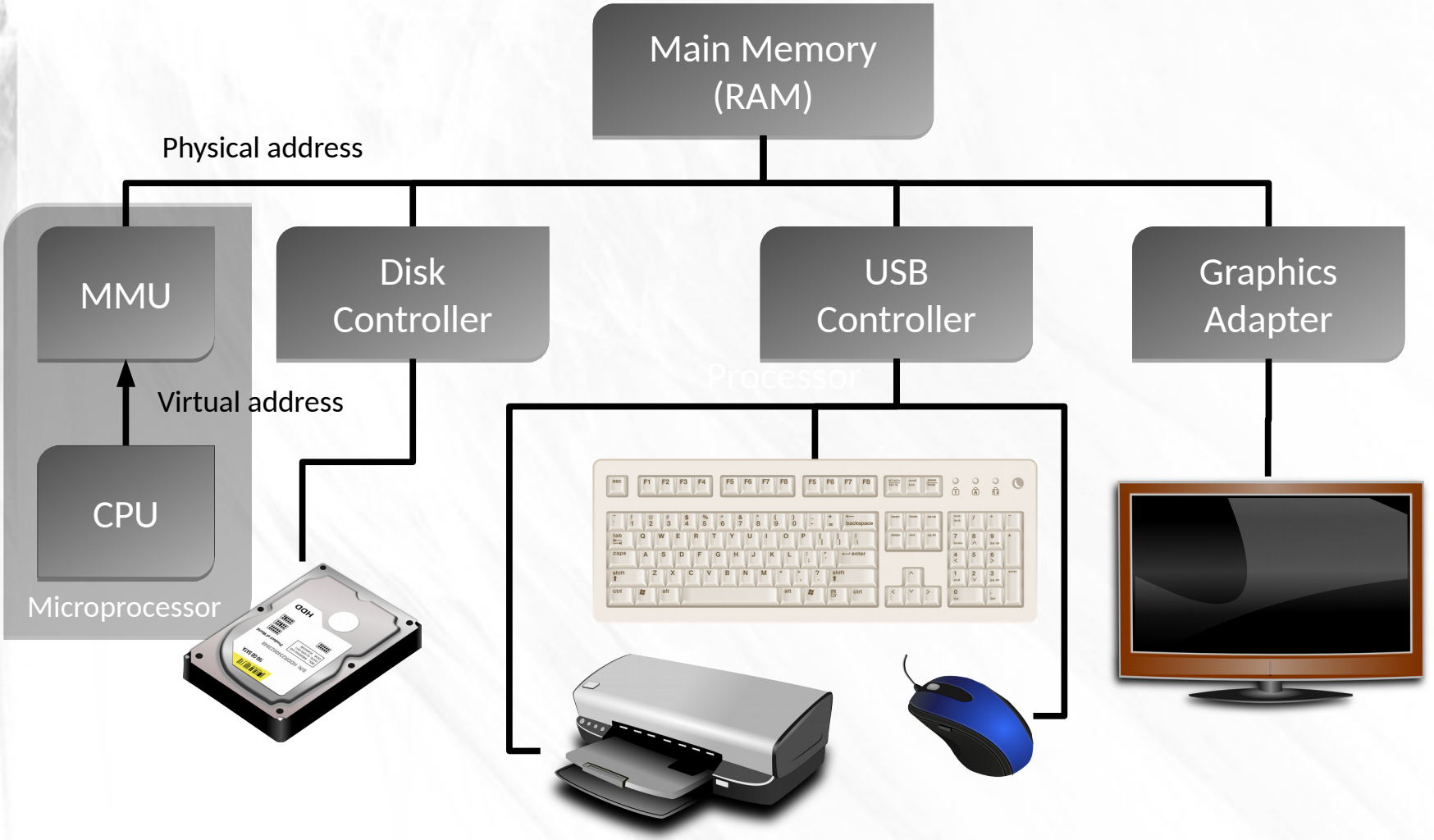
Unidad de Gestión de Memoria

- 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.

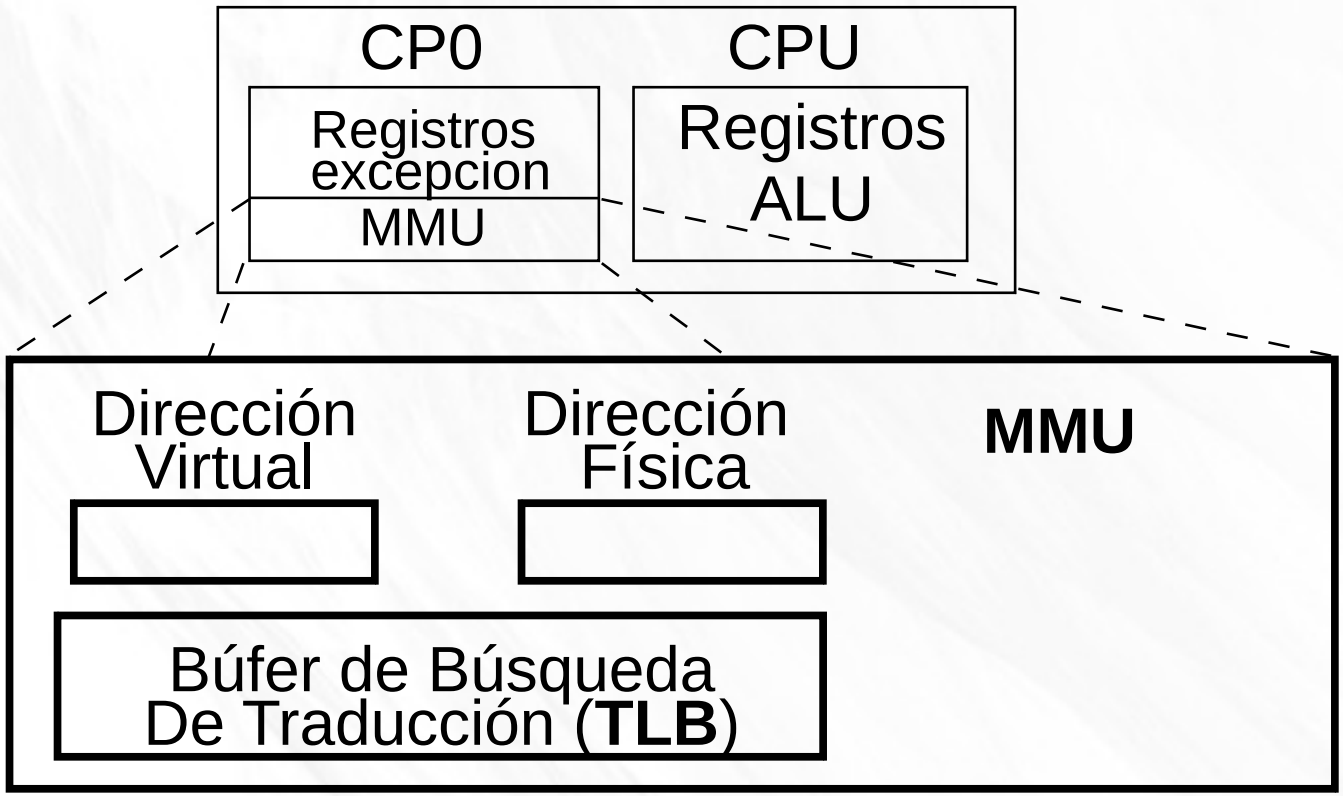
Unidad de Gestión de Memoria

- 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

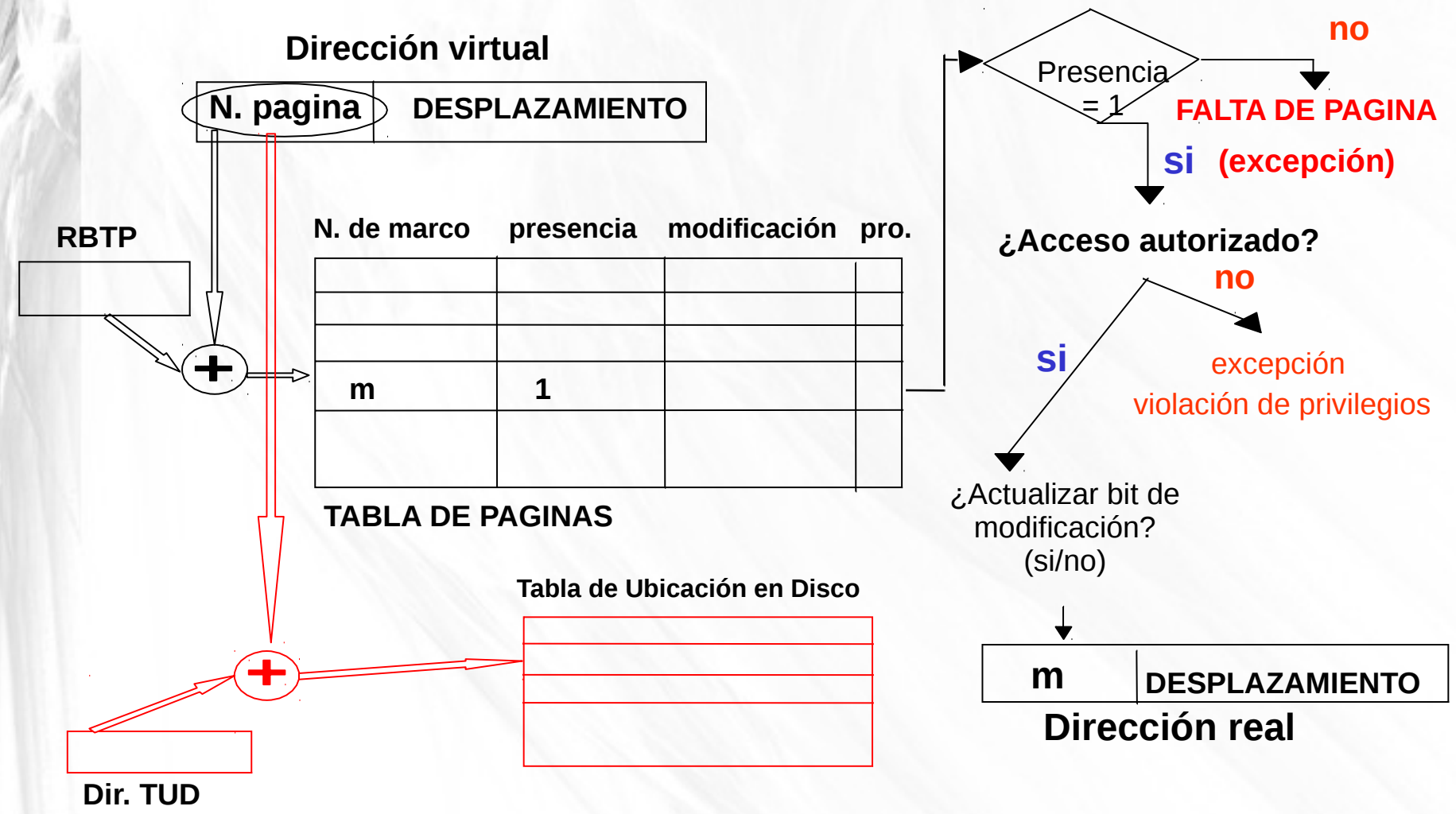
Memoria virtual mediante paginación por demanda (*demand paging*)

- 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) →

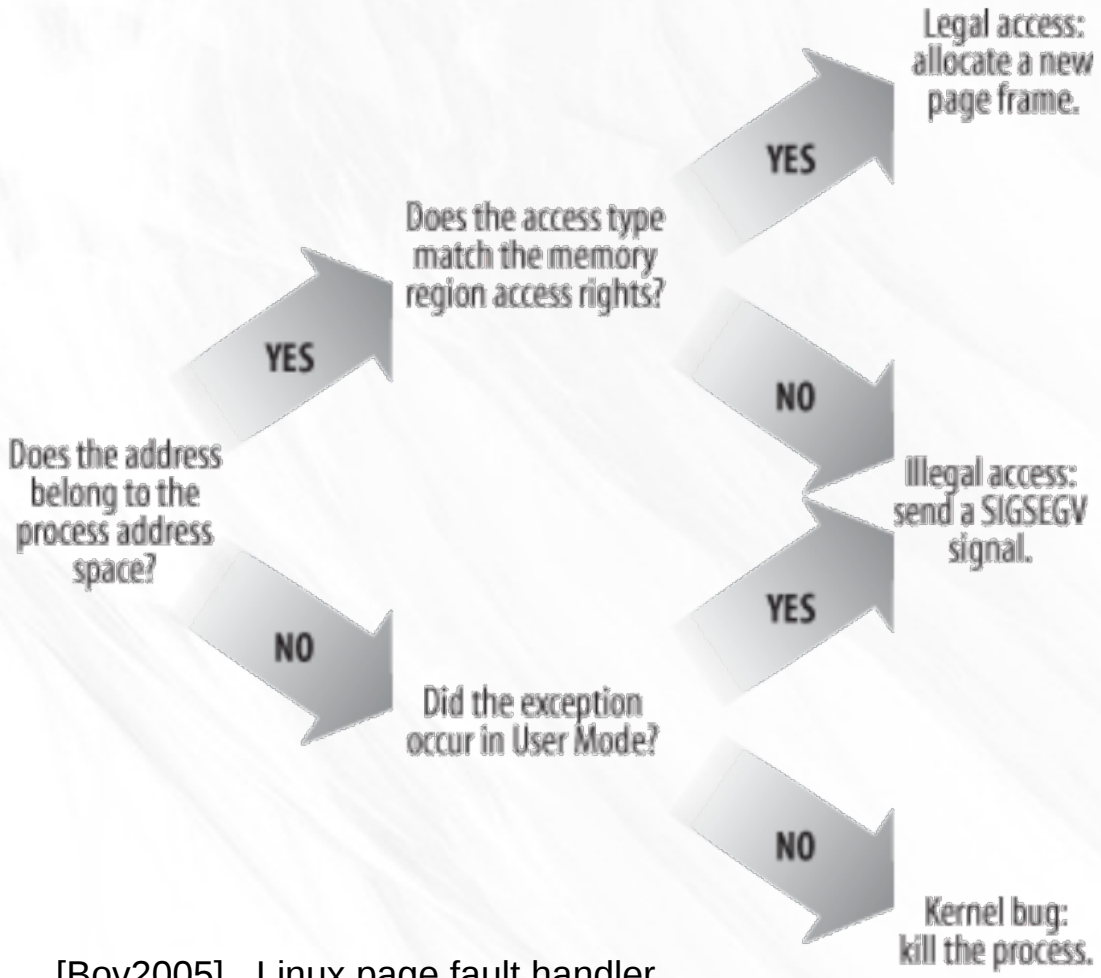
4.1. Actualizar TP(bit presencia=1, nº marco,...)

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	

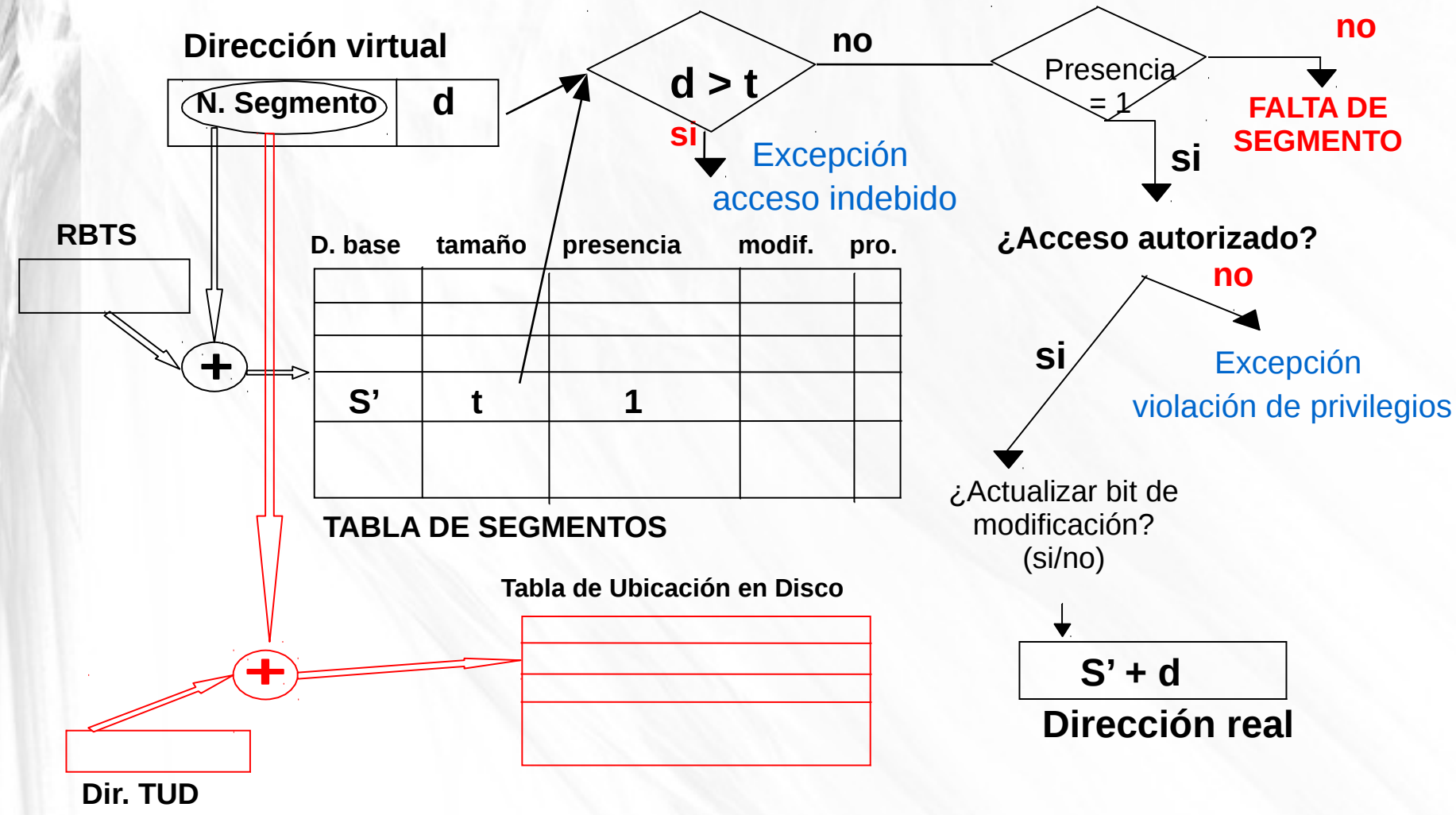
Compartición de páginas

- 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).

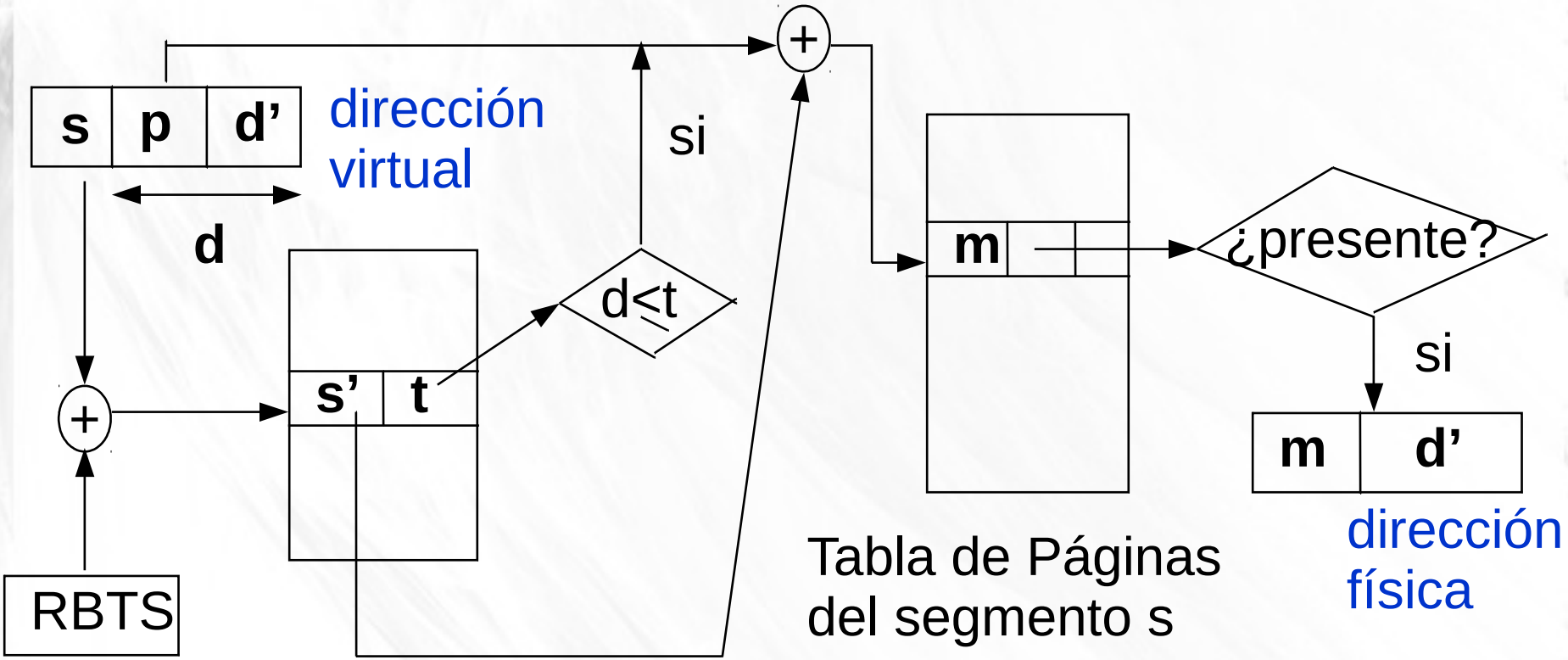
Esquema de traducción



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.

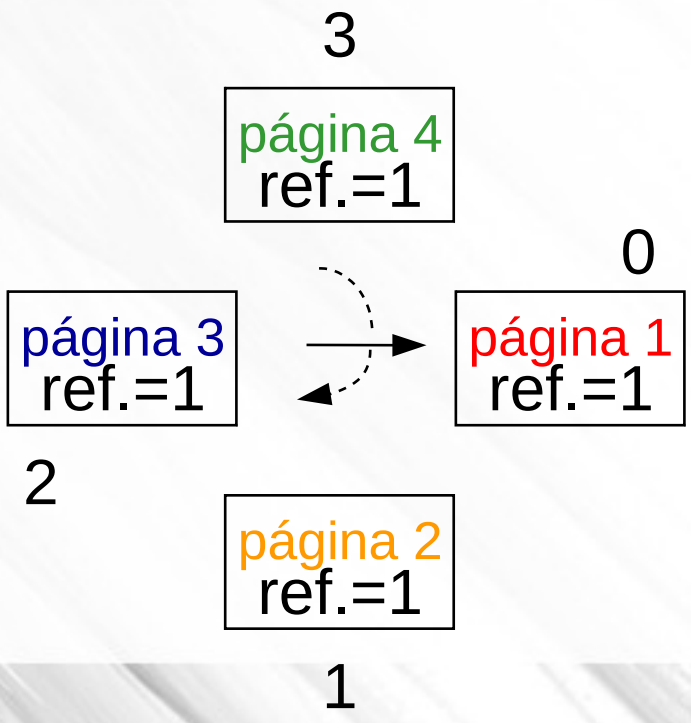
Algoritmo del reloj

- 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

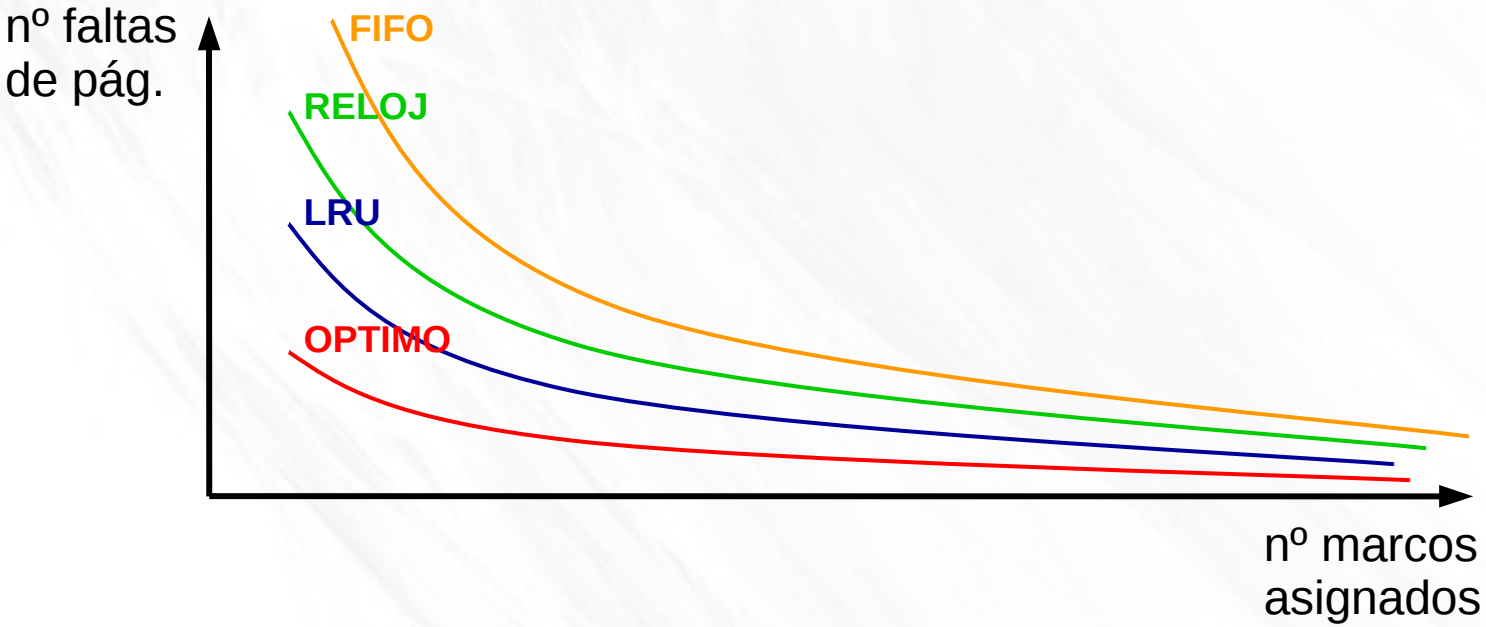
2. ¿Es R=0?

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.

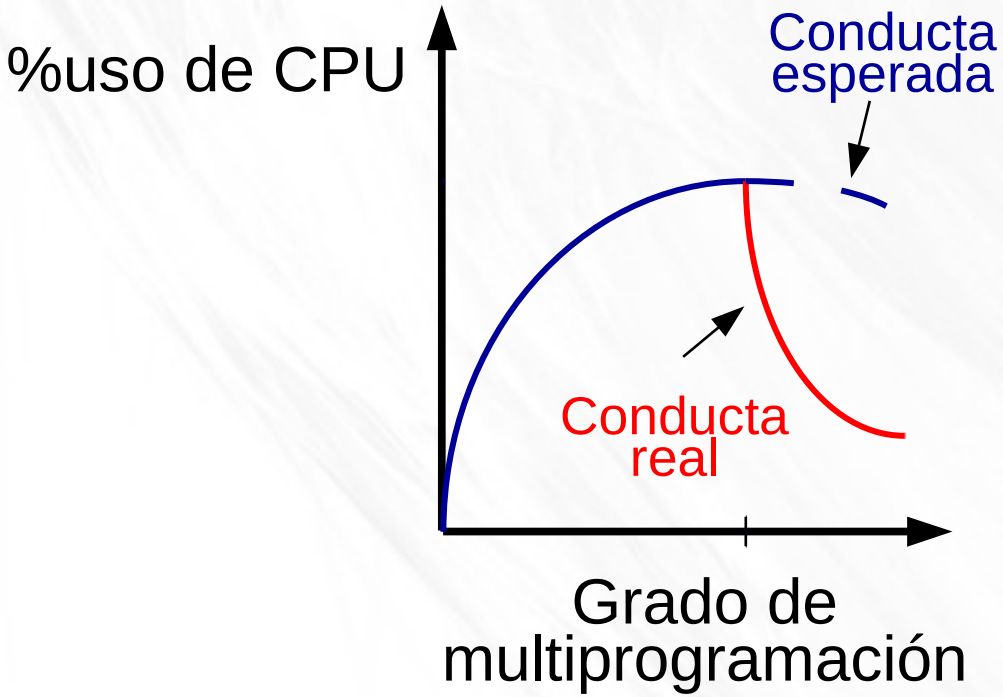


Hiperpaginación (*thrashing*)

- 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.



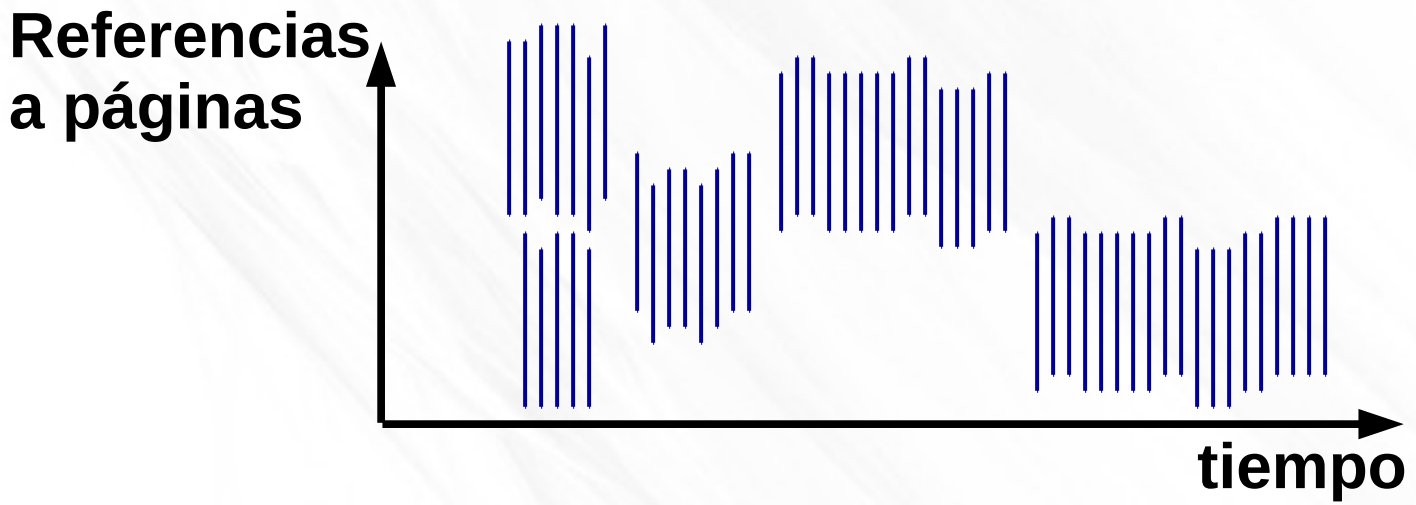
Hiperpaginación (*thrashing*)

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

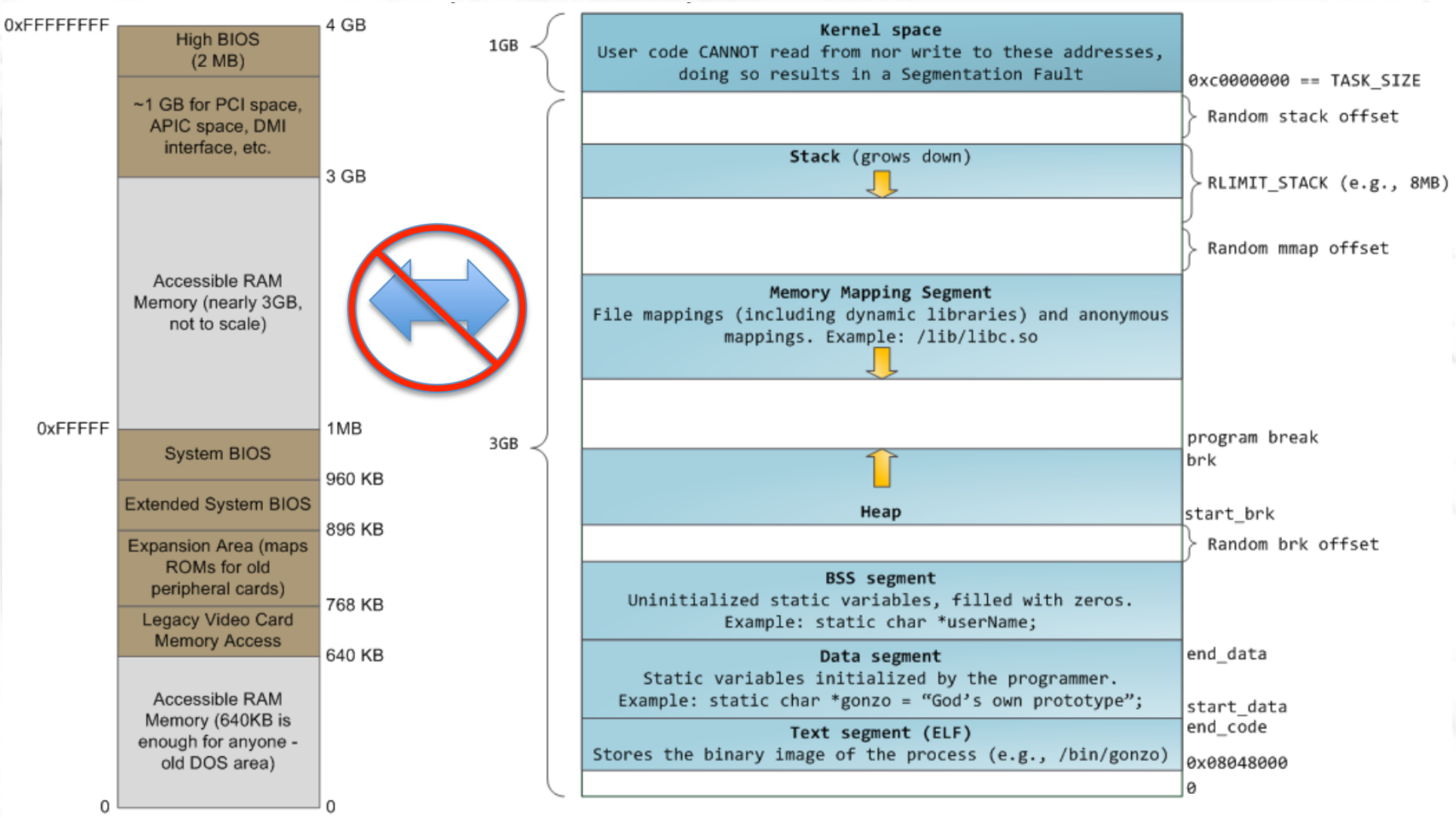
- La asignación y liberación de estructuras de datos es una de las operaciones más comunes en un kernel de SO. Para agilizar esta solicitud/liberación de memoria Linux usa el **nivel de bloques** (*slab layer*).
- El nivel de bloques actua como una caché de estructuras genérica.
  - Existe una caché 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

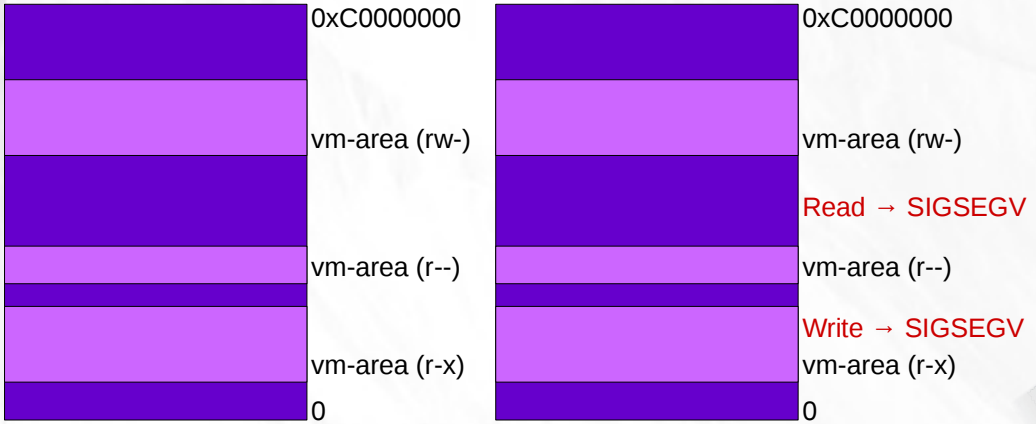


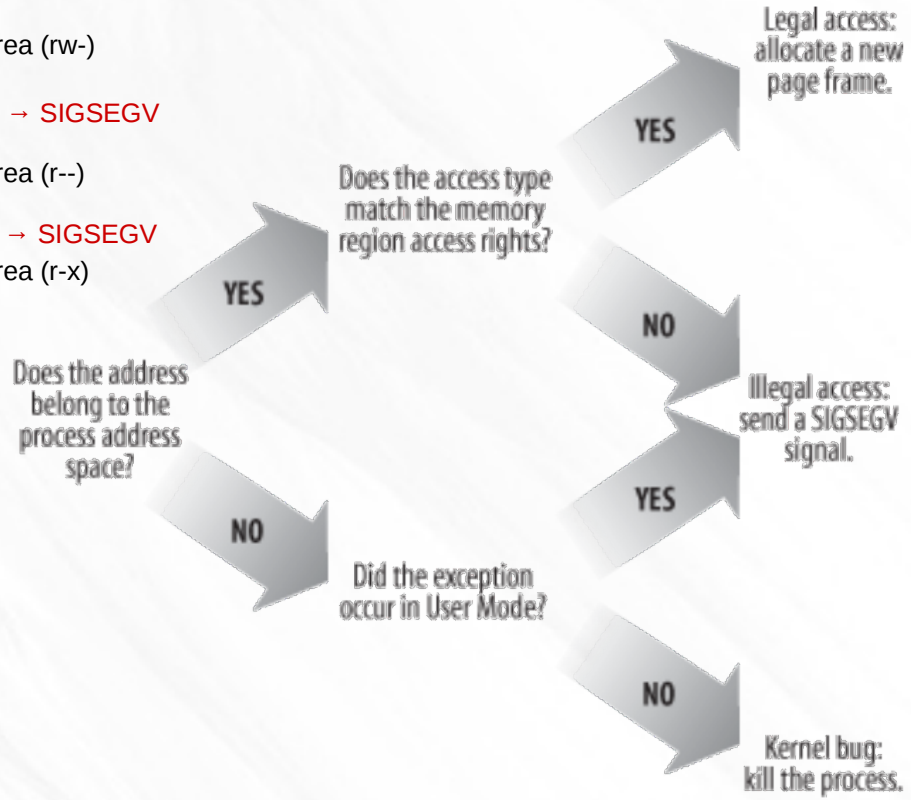
Espacio de direcciones de proceso

- 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?

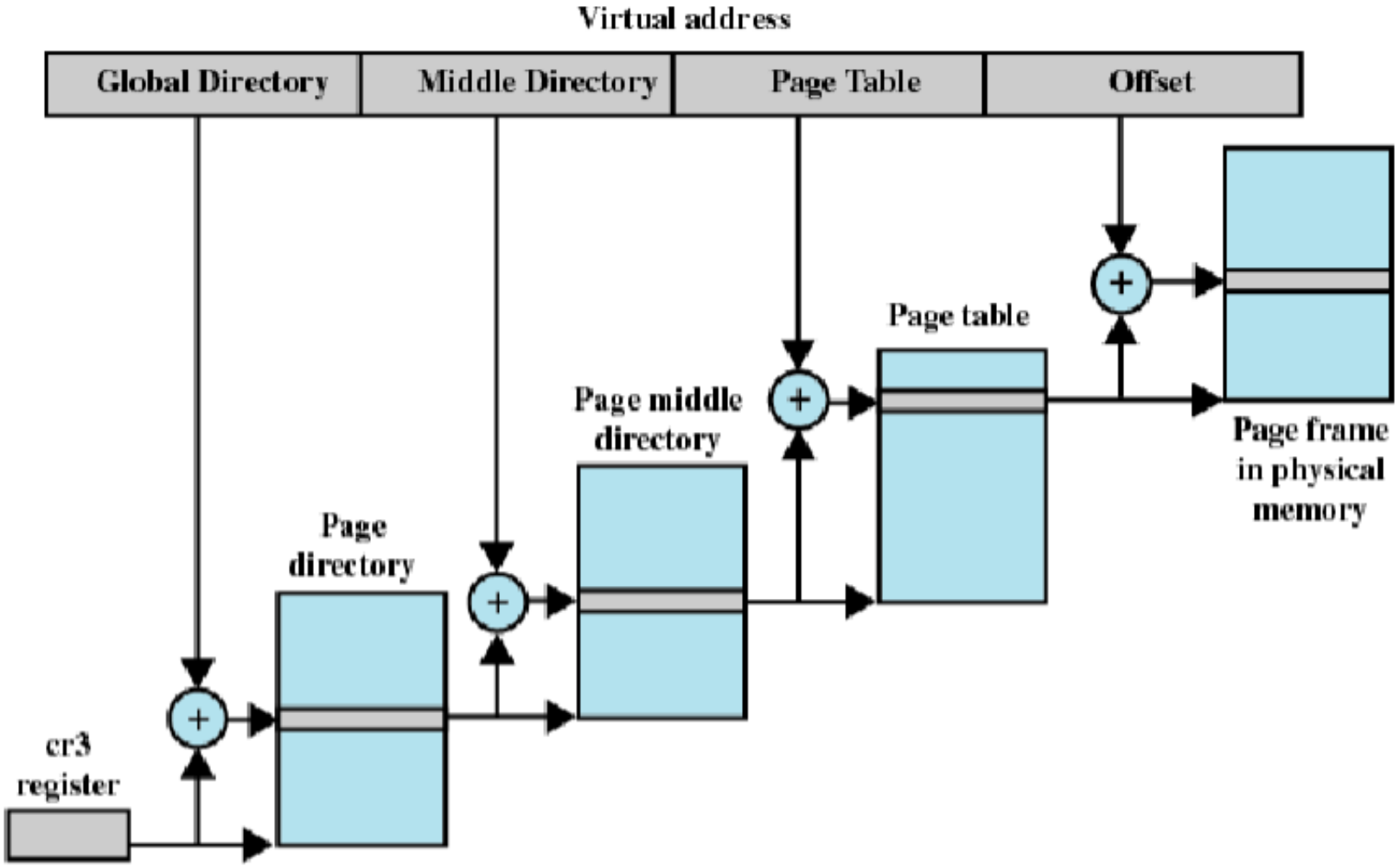
- `do_munmap()` permite eliminar un intervalo de direcciones. El parámetro `mm` especifica el descriptor de memoria (espacio de direcciones) del que se va a eliminar el intervalo de memoria que comienza en `start` y tiene una longitud de `len` bytes.

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

Tablas de páginas multinivel en Linux

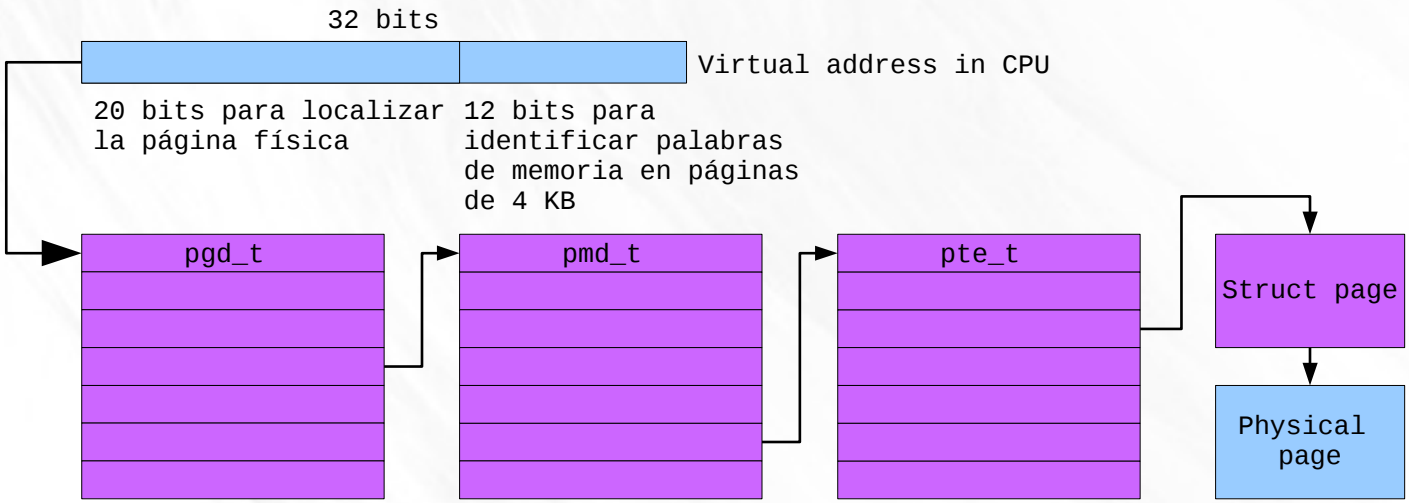
- Las direcciones virtuales deben convertirse a direcciones físicas mediante tablas de páginas. En linux tenemos 3 niveles de tablas de páginas.
  - La tabla de páginas de más alto nivel es el directorio global de páginas (del inglés page global directory, PGD), que consta de un array de tipo "pgd_t".
  - 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

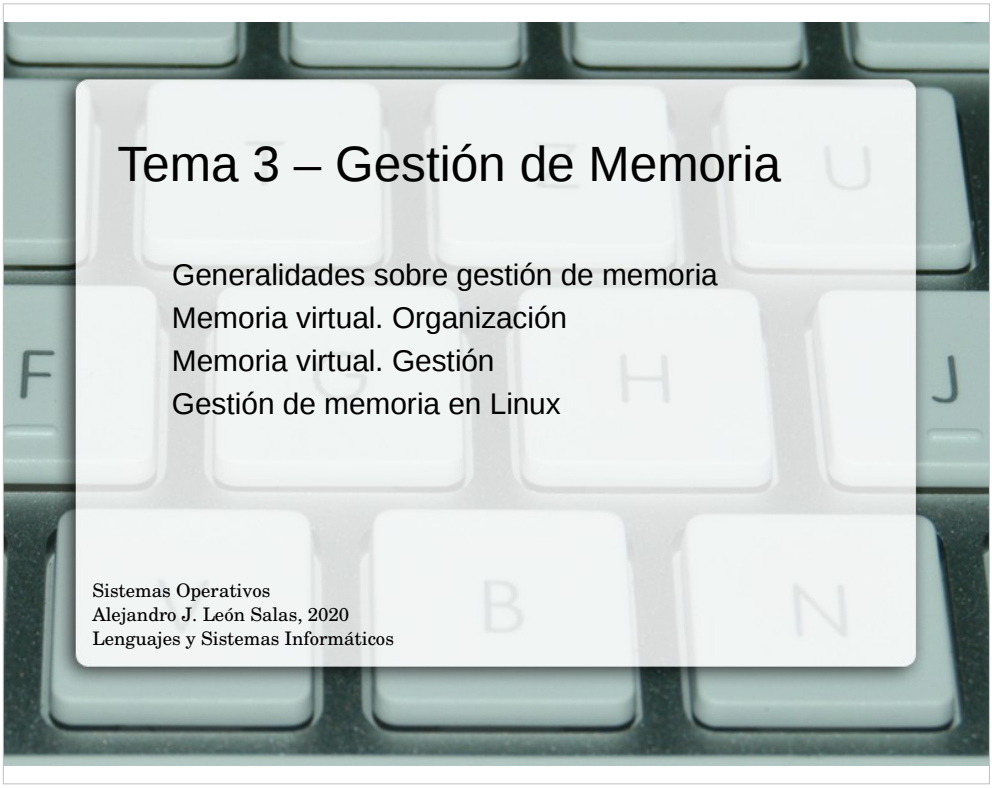
- Dos posibilidades dependiendo del objeto que representa la `struct address_space`:
- Si representa una proyección a memoria de un archivo, se activa el flag `PG_DIRTY` de la `struct_page` y ya está.
- Si representa un archivo, entonces se busca la página en la caché de páginas, y si no se encuentra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



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

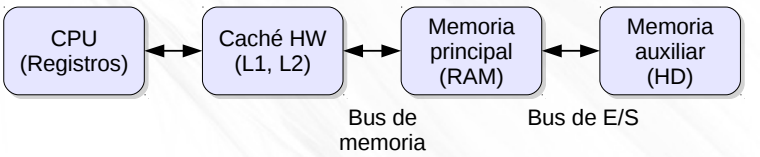
Generalidades sobre gestión de memoria

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

Comments.
Existen distintas formas de implementar los sistemas de memoria.
Idea: utilizar una jerarquía de memorias
Stalling pp.27-34, Silberschatz (p.35)

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.



Bus de memoria

Bus de E/S

Comments.
Existen distintas formas de implementar los sistemas de memoria.
Idea: utilizar una jerarquía de memorias
Stalling pp.27-34, Silberschatz (p.35)

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**.

Comments.
Stalling (capitulo 1 pp.31-34)

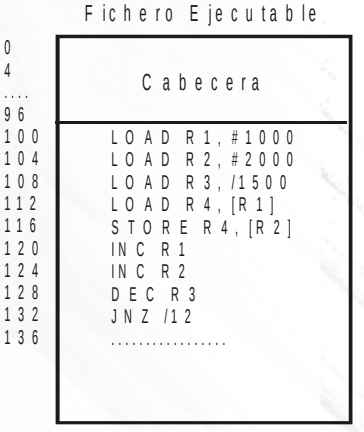
El uso de cachés es un principio importante en los sistemas de computación. La memoria principal puede considerarse como una caché rápida para la memoria secundaria.

TAE es el tiempo medio de acceso a una celda de memoria. El coste_acierto es el tiempo de acceso si hay acierto y el coste_fallo es el tiempo necesario en realizar el acceso en caso de fallo en la caché

Principio de localidad. Durante la ejecución de un programa las referencias a memoria por parte del procesador, tanto para instrucciones como para datos, tienden a estar agrupadas.

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

Código absoluto: las direcciones se generan en tiempo de compilación (y/o enlace)

* Hay que conocer las direcciones de memoria donde se va a ejecutar el programa en tiempo de compilación/enlace.
* El ejecutable no es reubicable.

Reubicación estática: las direcciones se generan en tiempo de carga (el ejecutable tiene referencias relativas)

* Una vez cargado no puede moverse a otro lugar de la memoria.
* Solo puede haber intercambio si el programa vuelven a la misma ubicación de memoria.

Reubicación dinámica: las direcciones se generan en tiempo de ejecución (el programa al ejecutarse maneja unas referencias que no son las direcciones de memoria reales a las que accede):

* No hay problema con el intercambio, los programas pueden salir de memoria y volver a ella en cualquier ubicación
* Aparece una distinción entre el espacio virtual o lógico de direcciones que maneja el programa y el espacio físico de direcciones al que realmente se accede.

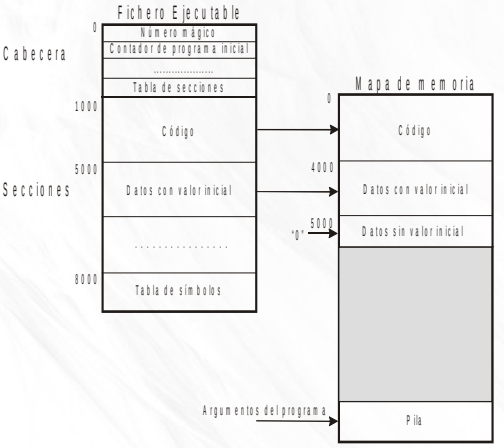
Normalmente, un programa reside en un disco como archivo binario ejecutable. Es preciso traer el programa a MP y asociarle un proceso para que se ejecute.

(Carretero) **Las direcciones lógicas son direcciones de memoria a las que hacen referencia las instrucciones de un programa. Las direcciones físicas son direcciones asignadas de memoria principal que se corresponden con las anteriores.**

En el ejemplo se muestra un archivo ejecutable cuyo contenido se ha puesto en un lenguaje ensamblador hipotético. Se ha supuesto que la cabecera ejecutable ocupa 100 bytes y que cada instrucción ocupa 4 bytes.

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

Comments.

(Carretero, pag. 172-182) **Formato del ejecutable**. Como parte final del proceso de compilación y montaje, se genera un archivo ejecutable que contiene el código del programa. En la figura se observa el formato típico de un ejecutable que está estructurado como un **conjunto de secciones**.

Cabecera: contiene información de control que permite interpretar el contenido del ejecutable:

- número mágico que identifica al ejecutable
- dirección del punto de entrada (*entry point*) del programa: se almacena inicialmente en el PC
- tabla que describe las secciones que aparecen en el ejecutable (tipo, dirección de comienzo en el archivo (offset) y tamaño)

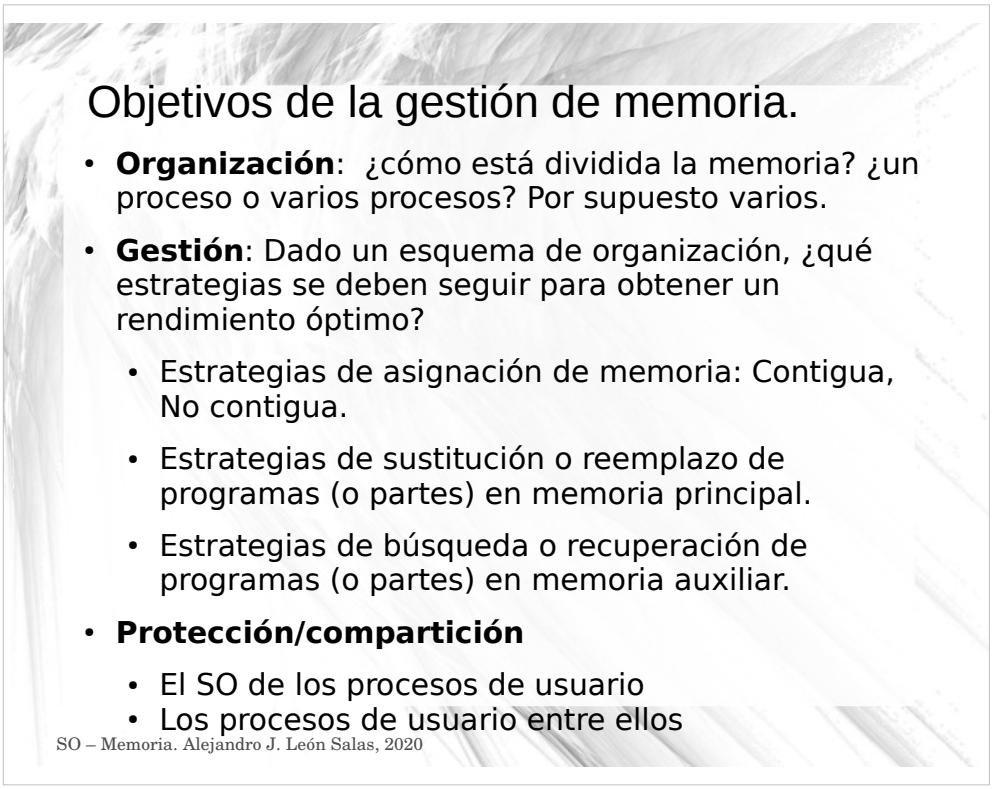
Tipos de secciones: (las más comunes)

- Código
- Datos con valor inicial (variables globales inicializadas en el programa)
- Datos sin valor inicial (variables globales sin inicializar) .bss – Block Started by Symbol

Las regiones que presenta el mapa de memoria inicial del proceso se corresponden básicamente con las secciones del ejecutable más la pila del proceso.

Los **sistemas operativos modernos** ofrecen un mapa de memoria dinámico en el que el mapa de un proceso está formado por un número variable de regiones que pueden añadirse o eliminarse durante su ejecución.

Otras regiones: **heap, archivos proyectados o mapeados, memoria compartida, pilas de threads**.



Comments.

Organización de la memoria: ¿un proceso o varios procesos? Si varios, ¿Cómo está dividida entre los distintos procesos?¿Puede tener un proceso parte de su espacio de direcciones en MP y parte en almacenamiento secundario? Todo depende del Hardware subyacente.

Gestión:

Estrategias de asignación (ubicación): ¿dónde situar un programa que no ha comenzado?

Estrategias de sustitución (reemplazo) ¿qué datos e instrucciones sacamos de memoria para meter otros?

Estrategias de búsqueda (recuperación): ¿cómo recuperar parte de un proceso que es necesario?

La parte del S.O que administra la memoria entre los distintos procesos se denomina **gestor de memoria**. Funciones del gestor de memoria:

a) Llevar el registro de las partes de la memoria ocupada y partes libres.
b) Saber que proceso ocupa cada parte utilizada.
c) Asignar la memoria a los nuevos procesos (asignación). Si no hay espacio libre estrategia de sustitución.
d) Administrar el intercambio memoria-disco, disco-memoria
e) Librar el espacio de los procesos que finalizan.
f) En memoria virtual estrategias de búsqueda

Conclusión: El rendimiento del sistema se ve afectado por el rendimiento de la gestión de memoria.

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

Comments.
Analizaremos cada esquema de gestión de memoria contigua respecto a los siguientes criterios:
* Memoria desaprovechada
* Tiempo dedicado a la asignación/liberación de memoria por los algoritmos

¿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.

Comments.

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.

Comments.
Problemas de E/S asíncronas sobre la memoria del proceso.
- En Unix solo está activo si la carga del sistema es muy grande.

Permite aumentar la proporción de procesos preparados -> mejora el uso del procesador
Existen ciertos tipos de procesos que no deberían ser intercambiados, por ejemplo, algunos procesos del sistema. Los SO que abordan este problema permiten que se especifique si es o no intercambiable pero es un privilegio restringido a una clase de procesos o de usuarios.

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

Comments.

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.

Comments.

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.

Comments.

Sin embargo, la velocidad de ejecución de un programa nunca es mayor que si se ejecuta dicho programa en un sistema sin MV, porque hay que acceder a las partes ausentes de su espacio de direcciones.

$ man 1 top

17. RES -- Resident Memory Size (KiB)

The non-swapped physical memory a task is using.

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.

Comments.

El MMU se dice que está gestionado por el SO en el sentido de que le suministra la información necesaria que cambia cuando se hace un cambio de contexto. A veces, también el SO se encarga de limpiar los registros del MMU: TLB (aunque a veces se hace por hardware en vez de por software).

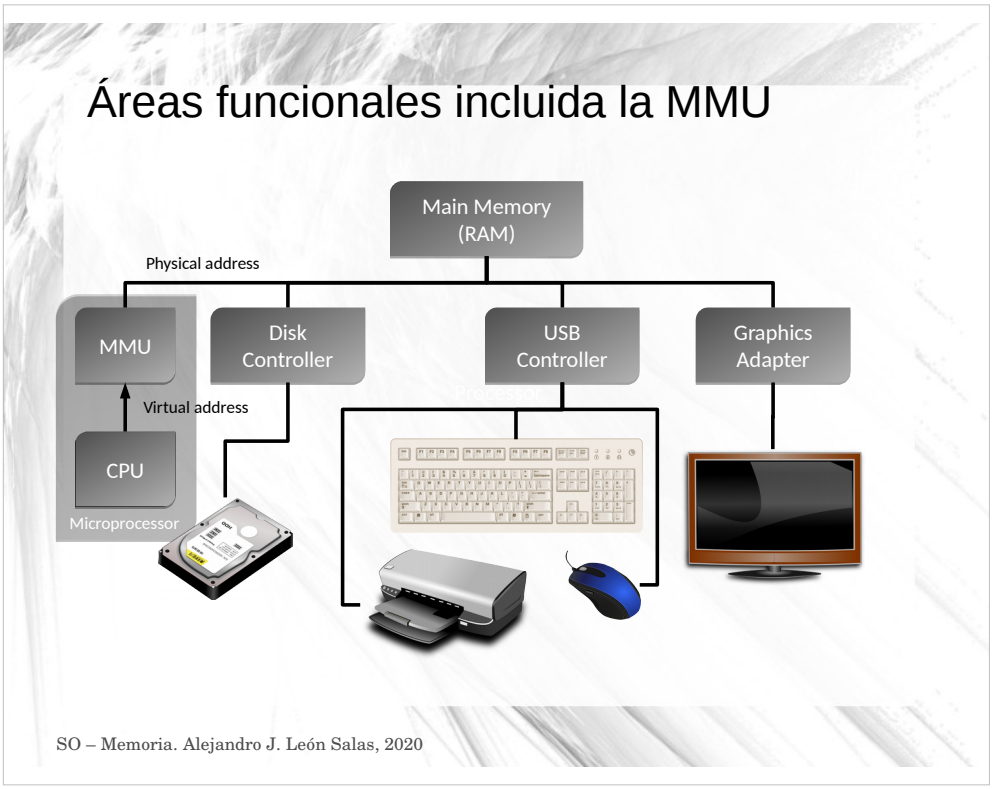
Unidad de Gestión de Memoria

- 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*).

Comments.

El TLB normalmente se traduce por bufer de traducción anticipada o adelantada.

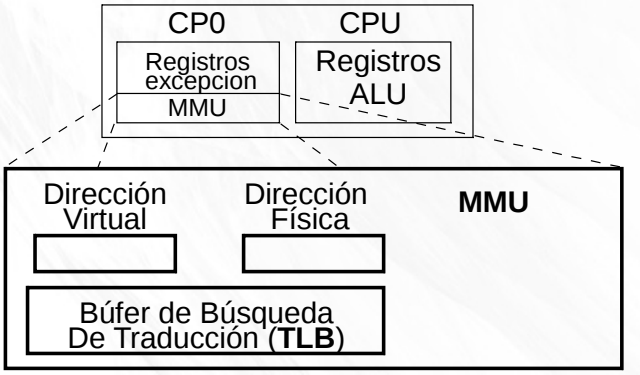
El MMU se dice que está gestionado por el SO en el sentido de que le suministra la información necesaria que cambia cuando se hace un cambio de contexto. A veces, también el SO se encarga de limpiar los registros del MMU: TLB (aunque a veces se hace por hardware en vez de por software).



Comments.

El MMU integrado en el esquema de areas funcionales genéricas de un sistema informático como se proporcionó en el tema 1.

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



Comments.

El MMU se dice que está gestionado por el SO en el sentido de que le suministra la información necesaria que cambia cuando se hace un cambio de contexto. A veces, también el SO se encarga de limpiar los registros del MMU: TLB (aunque a veces se hace por hardware en vez de por software).

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).

El tamaño del marco de página viene dado por el hardware y determina varios aspectos:

El número de bits menos significativos que se emplearán en las direcciones virtuales para completar direcciones físicas en el proceso de traducción. En otras palabras, el **offset en el marco de página y en la "página virtual"**.

El número de bits menos significativos de la dirección física alojada en cada entrada de la tabla de páginas que valdrán 0. **Bits menos significativos de la dirección física base del marco**.

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).

En el espacio virtual, una celda de memoria de granularidad 1 byte se identifica con la dirección virtual.

En el espacio físico, una celda de memoria de granularidad 1 byte se identifica mediante una dirección física (real), la cual se calcula mediante la dirección base de marco (almacenada en entrada TP y localizada con id_página_virtual) y el offset.

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.

La TP representa el mapa de memoria de la MV de un proceso. En un registro de la CPU se guarda la dirección de comienzo de la TP del proceso actual, PBTP. Su valor forma parte del PCB.

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

Comments.

Para ejecutar una instrucción o acceder a un dato, la página virtual que la/lo contiene debe estar presente en memoria principal.

El bit de validez/presencia tiene un doble significado:

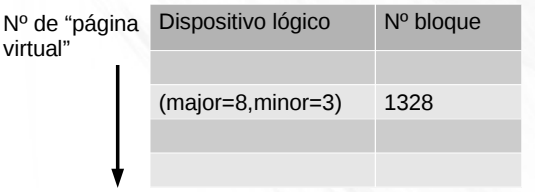
* Vale 1 si la página es válida y está cargada en MP
* Vale 0 si la página no está en MP (no está presente) o es una página inválida para el proceso, es decir, no pertenece a su espacio de direcciones.

Protección: leer, escribir, ejecutar

La TP está ordenada o indexada por número de “página virtual”, obtenido mediante los bits mas significativos de la dirección virtual del espacio de direcciones virtual del proceso.

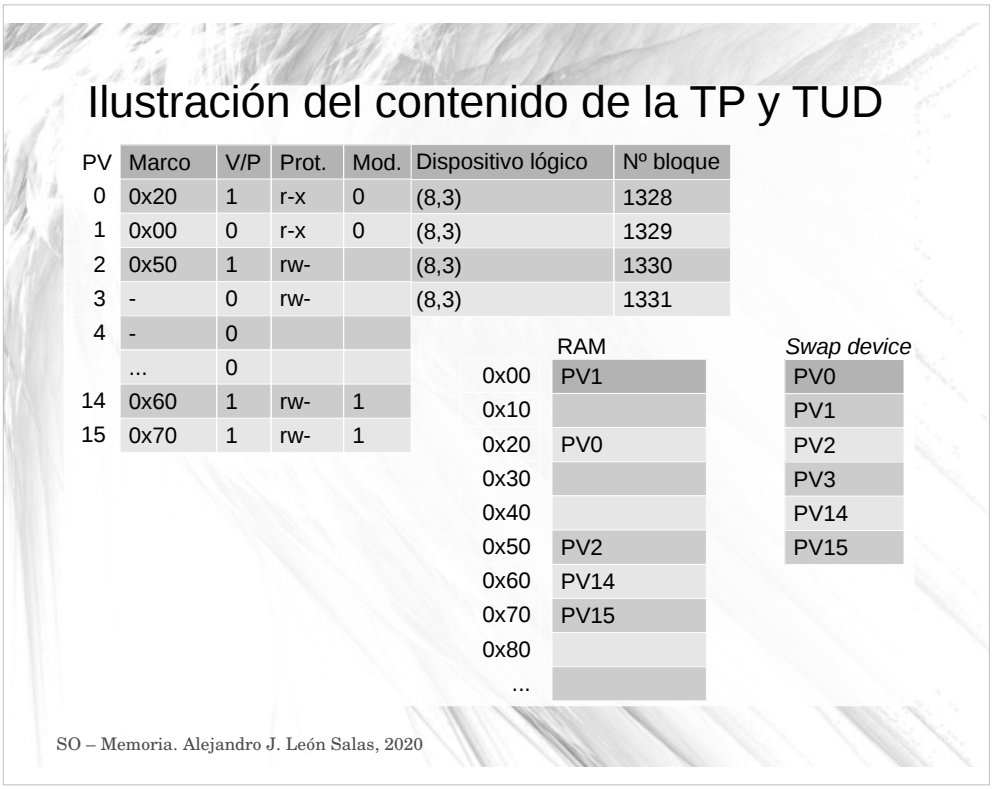
Tabla de Ubicación en Disco

- 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”.



Comments.

La TUD está ordenada o indexada por número de “página virtual”, obtenido mediante los bits mas significativos de la dirección virtual del espacio de direcciones virtual del proceso.



Comments.

El espacio virtual para este ejemplo es de 2^4 para páginas y 2^4 para desplazamiento. ¿PQ?

Las direcciones base de marco son de 8 bits, por lo que utlizo 8 bits para el bus de direcciones, con lo que en el espacio físico tengo 2^8 = 256 palabras de granularidad máxima 1Byte. Luego el espacio de memoria física tiene un tamaño de 256 palabras de 1 Byte.

La arquitectura divide la memoria en marcos de página de tamaño 16 bytes, y lo se porque los bit menos significativos de las direcciones base de marco que están a 0 son los 4 primeros, representados por la primera cifra hexadecimal.

Luego, el tamaño para una dirección virtual mínimo debe ser igual al offset en página física, 4 bits más el tamaño del resto de la dirección física (real) que son otros 4 bits. De esta manera podré utilizar toda la memoria física.

Sin embargo, podría ampliar el número de bits para identificar páginas virtuales.

Comentarios de algunas entradas:

* La entrada 3 pertenece a una página virtual válida que nunca se cargó en memoria pero que tiene respaldo.
* La entrada 4 no es válida porque no tiene respaldo.
* La entrada 1 estuvo en memoria en el marco 0x00 pero ahora no está.

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.

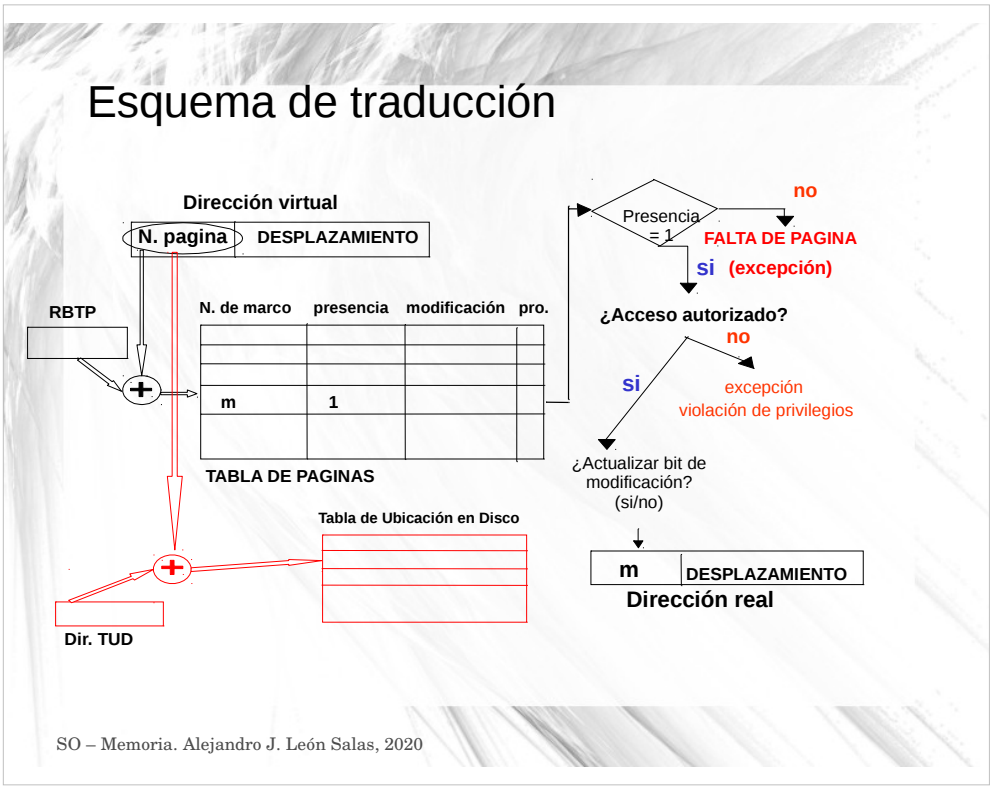
Comments.
Virtual memory can be implemented in:

- paging systems
- paged segmentation systems
- segmentation systems (However, segment replacement is much more sophisticated than page replacement since segments are of variable size.)

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*).

Comments.



Comments.

RBTP- registro base de la tabla de paginas (PTBR, del inglés *Page Table Base Pointer*): Información de memoria que se almacena en el PCB y se carga en un registro cuando el proceso está activo.

Antes de acceder mediante Nº. Página a la tabla de páginas se comprueba que el valor está dentro de los límites de la tabla mediante el Registro Límite de la tabla de páginas (PTLR, del inglés *Page Table Limit Register*).

https://www.geeksforgeeks.org/mapping-virtual-addresses-to-physical-addresses/

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.

Comments.

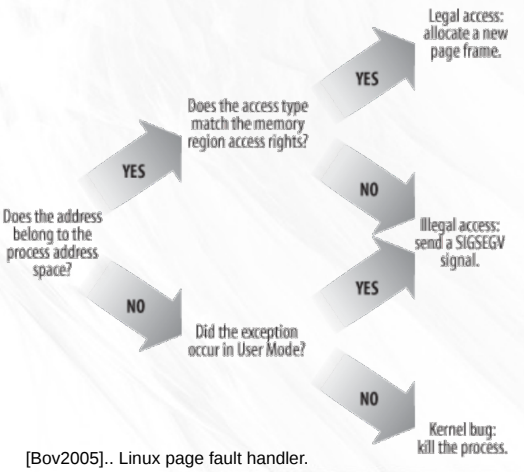
1) Trap the OS by exception.
2) Save registers and process state for the current process.
3) Check if the trap was caused because of a page fault and whether the page reference is legal.
4) If yes, determine the location of the required page on the backing store
5) Find a free frame [5].
6) Read (swap in) the required page from the backing store into the free frame. (During this I/O, the processor may be scheduled to some other process)
7) When I/O is completed, restore registers and process state for the process which caused the page fault and save state of the currently executing process.
8) Modify the corresponding page table entry to show that the recently copied page is now in memory.
9) Resume execution with the instruction that caused the page fault.

[5] If there is no free frames, the OS must choose a page in the memory (which is not the one being used) as a victim, and must swap it out (slow) to the backing store.

En el paso 5 se puede esperar por recurso del kernel MARCO LIBRE en lugar de aplicar política de sustitución (reemplazo) pero con el colchón de marcos de página no se da en la realidad.

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.

Comments.

As stated previously, **the Linux Page Fault exception handler must distinguish exceptions caused by programming errors from those caused by a reference to a page that legitimately belongs to the process address space but simply hasn't been allocated yet**.

The memory region descriptors allow the exception handler to perform its job quite efficiently. The do_page_fault( ) function, which is **the Page Fault interrupt service routine** for the 80×86 architecture, **compares the linear address that caused the Page Fault against the memory regions of the current process**; it can thus determine the proper way to handle the exception according to the scheme that is illustrated in Figure 9-4.

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.

Comments.

Un problema de la paginación es que se duplica el tiempo de acceso a memoria para la captación de una instrucción o dato, y por tanto, se reduce a la mitad el rendimiento del sistema.

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**.

Comments.

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.

Comments.

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

Comments.

Comments.

Supongo la memoria física de 4 GB dividida en 2G para kernel y 2G para user y por eso dir_k \in [0,2^31 – 1] y dir_u \in [2^31,2^32 – 1].

Por eso:

* espacio kernel empieza en dirección base de marco 0x00000000 y termina en 0x7FFFF000
* espacio user empieza en dirección base de marco 0x80000000 y termina en 0xFFFFF000

See on 0LEEME.txt

What does that amusing little vignette have to do with anything? In that world, it's easy to make decisions about how to divide up the 4 GiB address space you get from simple 32-bit addressing. Some OSes just split it in half, treating the top bit of the address as the "kernel flag": addresses 0 to 2^31-1 had the top bit cleared, and were for user space code, and addresses 2^31 through 2^32-1 had the top bit set, and were for the kernel. You could just look at the address and tell: 0x80000000 and up, it's kernel-space, otherwise it's user-space.

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`**.

Comments.
Google: linux what does ld.so mean
http://man7.org/linux/man-pages/man8/ld.so.8.html

The programs ld.so and ld-linux.so* find and load the shared objects (shared libraries) needed by a program, prepare the program to run, and then run it.

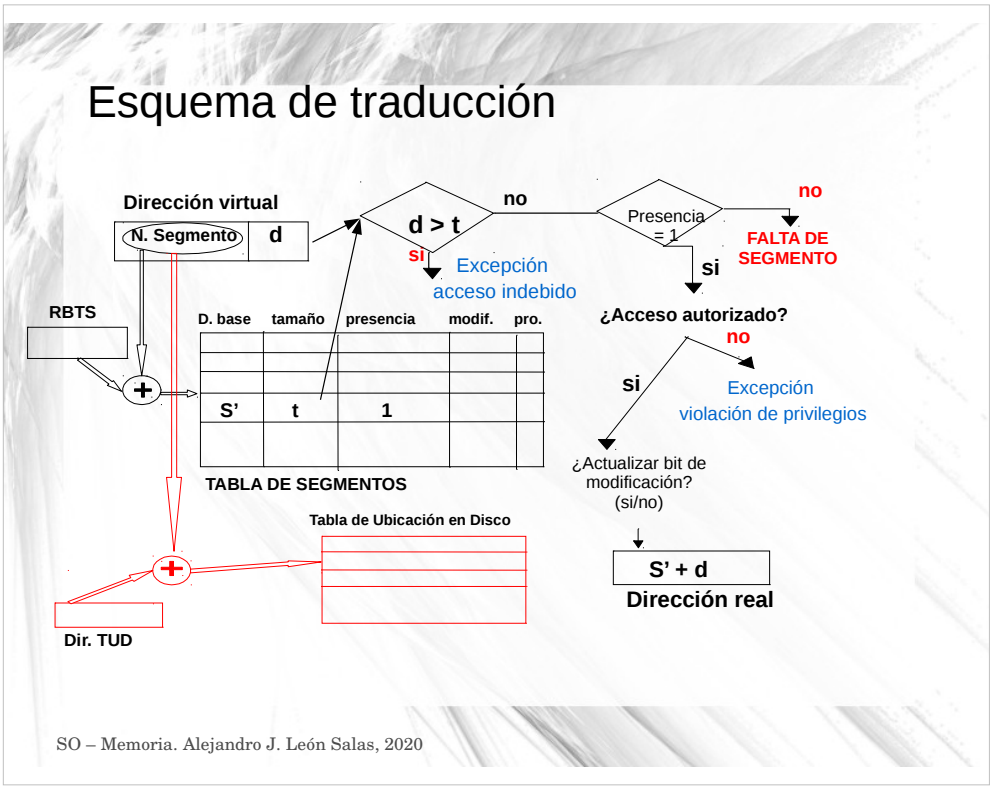
Memoria Virtual Segmentada

- **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).

Comments.

En el espacio virtual, una celda de memoria de granularidad 1 byte se identifica con la dirección virtual: (id_segment,offset).

En el espacio físico, una celda de memoria de granularidad 1 byte se identifica mediante una dirección física (real), la cual se calcula mediante la dirección base de segmento (almacenada en entrada TS y localizada con id_segmento) y el offset dentro del segmento.

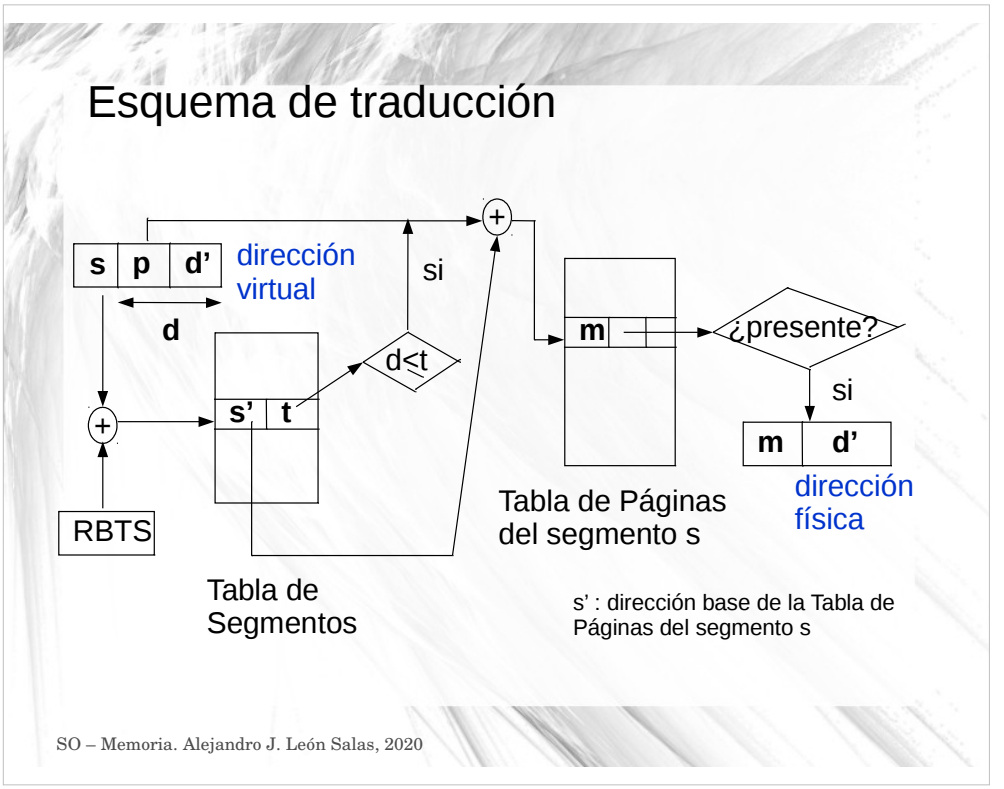


Comments.
D: desplazamiento
RBTS- Registro base de la tabla de segmentos
Información de memoria que se almacena en el PCB.
Se carga en un registro cuando el proceso está activo.
Antes de acceder mediante un Nº. Segmento a la tabla de segmentos se comprueba que el valor está dentro de los límites de la tabla (Registro Límite de la Tabla de Segmentos, RLTS)

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.

Comments.
* La compartición de código se hace a nivel de segmento ya que es el que tiene asociado el bit de protección.
* Características de paginación y segmentación
- Paginación: fragmentación interna.
- Segmentación: fragmentación externa y las estrategias de ubicación, asignación y liberación de memoria son más complejas.



Comments.

La TP representa el mapa de memoria de la MV de un proceso.En un registro de la CPU se guarda la dirección de comienzo de la TP del proceso actual, PBTP. Su valor forma parte del PCB.

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

Comments.

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

Comments.

*** Políticas de asignación**: espacio de MP asignado a un proceso. Asignar muchos marcos: reduce las faltas de página y el número de procesos activos. Asignar pocos marcos: aumenta la frecuencia de faltas de página y se degrada el tiempo de ejecución de los procesos.

*** Políticas de búsqueda**: cuándo llevar una página de disco a MP. Paginación por demanda: en cada falta de página. Desventaja de la anticipada: el camino de ejecución de un programa no es conocido a priori -> se pueden cargar páginas erróneas.

La paginación por demanda puede tener un efecto importante sobre el rendimiento del sistema. Para ver esto, podemos calcular el TAE (tiempo de acceso efectivo) de una memoria paginada por demanda. (siguiente trasparencia)

Las ventajas de la paginación por demanda son: Se garantiza que en MP solo están las páginas necesarias en cada momento y la sobrecarga de decidir qué páginas llevar a MP es mínima

*** Políticas de sustitución**: ¿Cuándo se carga una página en un marco ocupado?. El problema de la sustitución global es que el conjunto de páginas en MP de un proceso no va a depender sólo de su comportamiento (su actividad de paginación) sino también del resto de procesos.

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()`**).

Comments.

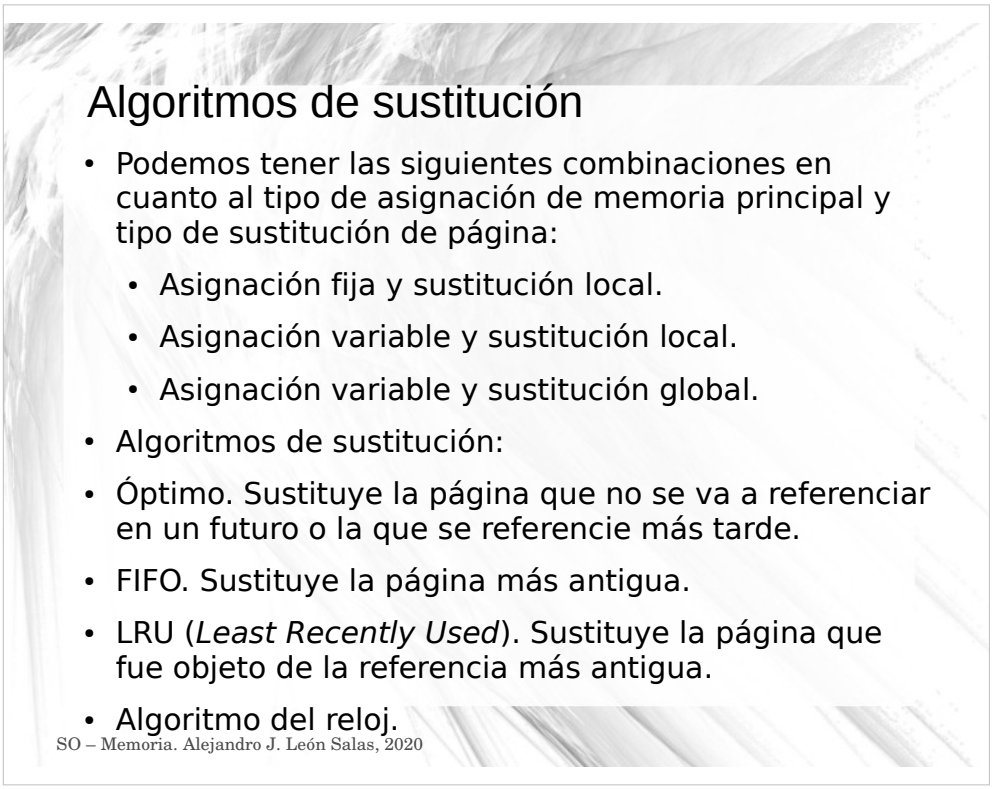
La TP representa el mapa de memoria de la MV de un proceso.En un registro de la CPU se guarda la dirección de comienzo de la TP del proceso actual, PBTP. Su valor forma parte del PCB.

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.

Comments.
Valores normales : de 512 bytes a 16 kb

* Aumento del tiempo de transferencia en función del tamaño de página por el tiempo que se tarda en leer/escribir una página desde/en disco. Podría ser necesario realizar varias E/S desde disco.
* Reduce la fragmentación interna: en promedio, la mitad de la última página de un proceso se desperdicia.
* Los tamaños de página siempre son potencia de 2. No existe un tamaño de página óptimo, existen distintos factores que apoyan distintos tamaños de página.



Comments.

Los algoritmos de sustitución están muy relacionados con las políticas de asignación de marcos de página a procesos:

* Asignación fija. Sustitución local : decidir por anticipado el número de marcos asignados.

* Asignación variable. Sustitución local : implementación sencilla

A page replacement algorithm determines how the victim page (the page to be replaced) is selected when a page fault occurs. **The aim is to minimize the page fault rate**.

The **efficiency** of a page replacement algorithm is evaluated by running it on a particular sequence (string) of memory references and computing the number of page faults.

Reference strings are either generated randomly, or by tracing the paging behavior of a system and recording the page number for each logical memory reference.

The **performance** of a page replacement algorithm is evaluated by running it on a particular string of memory references and computing the number of page faults.

Consecutive references to the same page may be reduced to a single reference, because they won't cause any page fault except possibly the first one:

(1,4,1,6,1,1,1,3) → (1,4,1,6,1,3).

We have to know the number of page frames available in order to be able to decide on the page replacement scheme of a particular reference string. Optionally, a frame allocation policy may be followed.

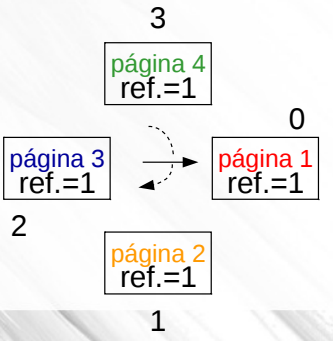
Algoritmo del reloj

- 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.



Comments.

Aproximación o variante menos costosa que LRU.

Ejemplo: cadena B E B A B C D

1 A R=1

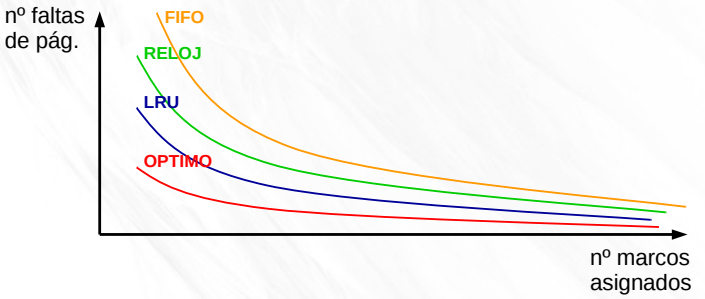
2 B R=1

3 C R=1

4 D R=1

Comparativa de algoritmos

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



Comments.

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.

Comments.

A process is thrashing if it is spending more time for paging in/out (due to frequent page faults) than executing.

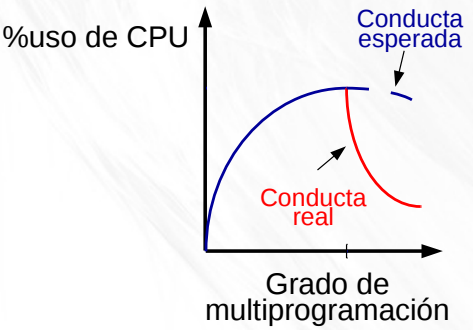
Often a heavily loaded computer has so many processes queued up that, if all the processes were allowed to run for one scheduling time slice, they would refer to more pages than there is RAM, causing the computer to "thrash".

By swapping some processes from memory, the result is that processes—even processes that were temporarily removed from memory—finish much sooner than they would if the computer attempted to run them all at once. The processes also finish much sooner than they would if the computer only ran one process at a time to completion, since it allows other processes to run and make progress during times that one process is waiting on the hard drive or some other global resource.

In other words, the working set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible. Thus it optimizes CPU utilization and throughput.

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.



Comments.

Thrashing causes considerable degradation in system performance. If a process does not have enough number of frames allocated to it, it will issue a page fault. A victim page must be chosen, but the problem is that all pages are in active use. So, the victim page selection will cause a new page replacement will be needed to be done in a very short time. This means another page fault will be issued shortly, and so on and so forth.

In case a process thrashes, the best thing to do is to suspend its execution and page out all its pages in the memory to the backing store.

Local replacement algorithms can limit the effects of thrashing. If the degree of multiprogramming is increased over a limit, processor utilization falls down considerably because of thrashing.

To prevent thrashing, we must provide a process as many frames as it needs. For this, a model called the working set model is developed which depends on the locality model of program execution.

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.

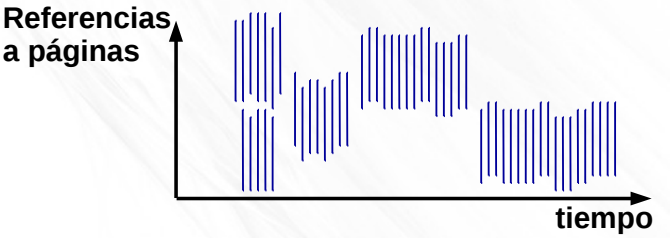
Comments.

Los **algoritmos de regulación de carga** intentan detectar el comienzo de la hiperpaginación y mantener el grado de multiprogramación próximo a un valor óptimo.

Ejemplo: criterio del 50% (si el **tiempo del núcleo** sobrepasa el 50% del tiempo total de CPU, es que se está produciendo la hiperpaginación.

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.).



Comments.
Se pretende que el número de faltas de página sea mínimo.
Podemos observar el comportamiento de los programas en tiempo de ejecución para llegar a ciertas conclusiones que nos serán útiles para diseñar los algoritmos de sustitución.

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)

Comments.

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

Comments.
No podemos saber a priori qué páginas se referenciarán y cuales no.
Es una propiedad empírica.
Definición. **Localidad**. Es un pequeño grupo de páginas, no necesariamente adyacentes, que son referenciadas durante un periodo de tiempo.

Normalmente el paso de una localidad a otra es lento.

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.

Comments.

Peter Denning (1968) defines “the working set of information W (t , τ) of a process at time t to be the collection of information referenced by the process during the process time interval (t − τ , t)”. Typically the units of information in question are considered to be memory pages. This is suggested to be an approximation of the set of pages that the process will access in the future (say during the next τ {\displaystyle \tau } \tau time units), and more specifically is suggested to be an indication of what pages ought to be kept in main memory to allow most progress to be made in the execution of that process.

Rationale

The effect of choice of what pages to be kept in main memory (as distinct from being paged out to auxiliary storage) is important: if too many pages of a process are kept in main memory, then fewer other processes can be ready at any one time. If too few pages of a process are kept in main memory, then the page fault frequency is greatly increased and the number of active (non-suspended) processes currently executing in the system approaches zero.

The working set model states that a process can be in RAM if and only if all of the pages that it is currently using (often approximated by the most recently used pages) can be in RAM. The model is an all or nothing model, meaning if the pages it needs to use increases, and there is no room in RAM, the process is swapped out of memory to free the memory for other processes to use.

Often a heavily loaded computer has so many processes queued up that, if all the processes were allowed to run for one scheduling time slice, they would refer to more pages than there is RAM, causing the computer to "thrash".

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.

Comments.

Peter Denning (1968) defines “the working set of information W (t , τ) of a process at time t to be the collection of information referenced by the process during the process time interval (t − τ , t)”. Typically the units of information in question are considered to be memory pages. This is suggested to be an approximation of the set of pages that the process will access in the future (say during the next τ {\displaystyle \tau } \tau time units), and more specifically is suggested to be an indication of what pages ought to be kept in main memory to allow most progress to be made in the execution of that process.

Rationale

The effect of choice of what pages to be kept in main memory (as distinct from being paged out to auxiliary storage) is important: if too many pages of a process are kept in main memory, then fewer other processes can be ready at any one time. If too few pages of a process are kept in main memory, then the page fault frequency is greatly increased and the number of active (non-suspended) processes currently executing in the system approaches zero.

The working set model states that a process can be in RAM if and only if all of the pages that it is currently using (often approximated by the most recently used pages) can be in RAM. The model is an all or nothing model, meaning if the pages it needs to use increases, and there is no room in RAM, the process is swapped out of memory to free the memory for other processes to use.

Often a heavily loaded computer has so many processes queued up that, if all the processes were allowed to run for one scheduling time slice, they would refer to more pages than there is RAM, causing the computer to "thrash".

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	
	*		*	*	*			*	*	

SO – Memoria. Alejandro J. León Salas, 2020

Comments.
Como desventajas tenemos la estimación de la ventana del conjunto de trabajo, \tau, y el gasto de tiempo de núcleo en cada referencia.

The main hurdle in implementing the working set model is keeping track of the working set. The working set window is a moving window. At each memory reference a new reference appears at one end and the oldest reference drops off the other end. A page is in the working set if it is referenced in the working set window.

To avoid the overhead of keeping a list of the last k referenced pages, the working set is often implemented by keeping track of the time t of the last reference, and considering the working set to be all pages referenced within a certain period of time.

The working set isn't a page replacement algorithm, but page-replacement algorithms can be designed to only remove pages that aren't in the working set for a particular process. One example is a modified version of the clock algorithm called WSClock.

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.

Comments.

La TP representa el mapa de memoria de la MV de un proceso.En un registro de la CPU se guarda la dirección de comienzo de la TP del proceso actual, PBTP. Su valor forma parte del PCB.

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

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

Comments.
El intervalo es cerrado [t_{c-1}, t_c]

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
*	*			*	*			*	*	

Comments.

Problema (Stalling): no rinde muy bien en los periodos de transición entre dos WS -> ninguna página se retira del WS hasta que ocurra una nueva f.p. -> todos los WS se inflan y esto puede provocar activaciones/desactivaciones de procesos -> Podemos tener muchas páginas innecesarias en MP mientras no se dé una f.p.

Cuestión (Silberschatz, 314) ¿existe un mínimo número de marcos de página a asignar a un proceso? Respuesta: Si, depende de la arquitectura del computador y de sus instrucciones. El proceso siempre debe disponer del número de marcos necesarios para poder ejecutar una instrucción máquina. Supongamos que tienes una referencia a memoria -> necesitamos dos marcos. Si se permite indirección a un nivel, entonces necesitamos 3 marcos.

¿Y debe existir un máximo? el número máximo de marcos de MP.

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`**.

Each of these pages is given a unique number; the **page frame number (PFN)**.

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;
…
}
```


Comments

El campo flags almacena el estado de la página. Por ejemplo si la página ha sido modificada (PG_dirty) o está bloqueada en memoria (PG_locked).

El campo _count mantiene el número de referencias a la página. El kernel usa la función page_count() para saber si una página está libre (devuelve 0) o está siendo usada (devuelve un valor mayor que 0). (Esto implementa la lista de marcos de página (pfdata))

El campo virtual mantiene la dirección de la página en memoria virtual.

El kernel utiliza la estructura struct page para gestionar todas las páginas físicas, ya que necesita saber si una página está libre o asignada. Si la página no está libre el kernel necesita saber quién está utilizando la página.

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.

Comments.

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

Debido a las limitaciones HW, el kernel no puede tratar a todas las páginas físicas de igual forma. Algunas páginas no pueden usarse para determinadas tareas, pej. La zona reservada para acceso DMA. El kernel clasifica las páginas en zonas. (https://www.kernel.org/doc/gorman/html/understand/understand005.html)

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.

Los **modificadores de acción** permiten especificar como va a asignar el kernel la memoria solicitada. Por ejemplo, __GFP_WAIT permite que el que solicita la asignación de memoria pueda entrar en estado sleep y __GFP_IO permite que el que solicita la asignación de memoria pueda comenzar entrada salida de disco.

Los **modificadores de zona** especifican de que zona asignar la memoria. Linux permite dividir el espacio de memoria en zonas dependientes de la arquitectura utilizada. Por ejemplo, en la arquitectura x86 ZONE_NORMAL permite asignar memoria física en el rango 16-896 MB, mientras que ZONE_DMA permite asignar memoria inferior a 16MB que es la única que pueden utilizar los dispositivos DMA en esta arquitectura. Un ejemplo de uso de este flag sería __GFP _DMA el cual indica al kernel que asigne memoria solamente de la zona reservada para DMA.

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.

Comments.

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.

Comments.

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);
```


Comments.

La función kmalloc() es similar a la que se utiliza en espacio de usuario para solicitar memoria: malloc(), con la diferencia del parámetro flags. Se utiliza para solicitar al kernel un trozo de memoria en bytes. La región de memoria física asignada tras la ejecución de la rutina es contigua.

Si la llamada a kmalloc tiene éxito, p apunta a un bloque de memoria con un tamaño que al menos tiene el tamaño requerido por el primer parámetro: size, si falla devuelve NULL.

kfree() libera un bloque de memoria previamente asignado mediante kmalloc().

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’

*/
```


Comments.

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);
```


¿Cómo se puede solicitar memoria en el espacio virtual en lugar de directamente en el espacio físico?
vmalloc() pag. 244

La función **vmalloc()** permite asignar memoria que es solamente contigua en el espacio virtual pero no es necesariamente contigua en la memoria física. **Así es como trabaja la función malloc(), la cual garantiza que las páginas devueltas son contiguas en el espacio virtual del procesador pero no en el espacio físico de la RAM.** El kernel hace que las páginas que potencialmente no son contiguas en RAM lo sean en el espacio virtual rellenando adecuadamente las tablas de páginas.

A pesar del hecho de que solo en ciertos casos es necesario disponer de memoria física contigua, la mayoría del código kernel usa kmalloc() en lugar de vmalloc() para obtener memoria. El principal motivo es para incrementar el rendimiento. La función vmalloc() debe rellenar explícitamente las entradas de las tablas de páginas para lograr hacer contiguas en espacio virtual las páginas físicas, lo que implica que cada página física debe proyectarse en su página virtual, lo que a su vez implica un aumento en el consumo de entradas del TLB con respecto a cuando se proyecta la memoria directamente como en el caso de kmalloc().

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^{order}).
  - Cada bloque (*slab*) contiene estructuras de su tipo.

Comments

El slab layer clasifica los diferentes objetos (estructuras) en grupos llamados cachés, cada una de las cuales almacena un tipo de objeto distinto, e.d. existe una caché por tipo de objeto.

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.

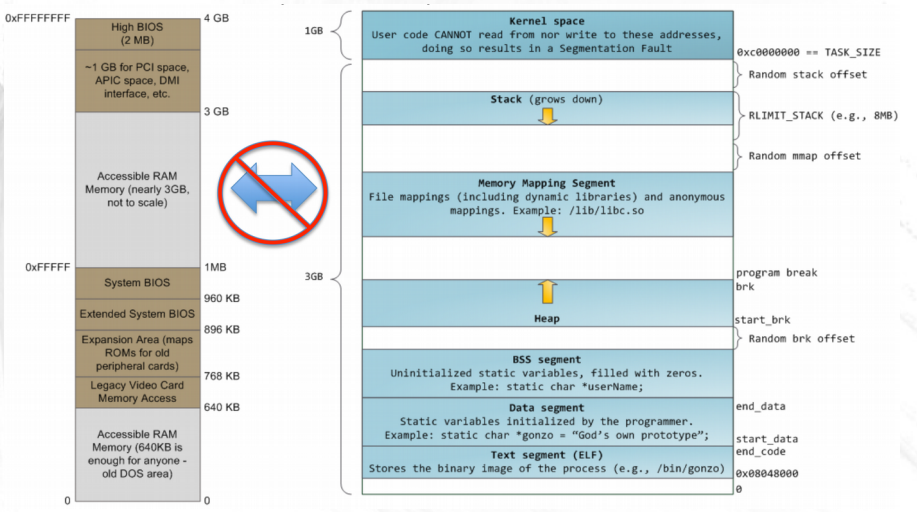
```
p = kmalloc(sizeof(struct task_struct),GFP_KERNEL);
```


La estrategia de asignación de estructuras mediante bloques llenos, parcialmente llenos y vacíos reduce la fragmentación de los bloques.

El objetivo que permite alcanzar esta caché de estructuras es reducir la asignación/liberación de páginas físicas. De hecho, el slab layer solo invoca la función de asignación de páginas cuando no existen bloques parciales ni vacíos en una determinada caché. La función de liberación de páginas solamente se llama cuando la memoria del sistema disponible cae por debajo de un umbral o la caché se destruye explícitamente.

Kmalloc() y kfree() actúan por encima del nivel de bloques.

Espacio de direcciones de proceso



SO – Memoria. Alejandro J. León Salas, 2020

Comments.
X86 address space vs. process address space in Linux.

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)**.

El proceso mediante llamadas al kernel puede dinámicamente añadir y eliminar áreas de memoria a/de su espacio de direcciones.

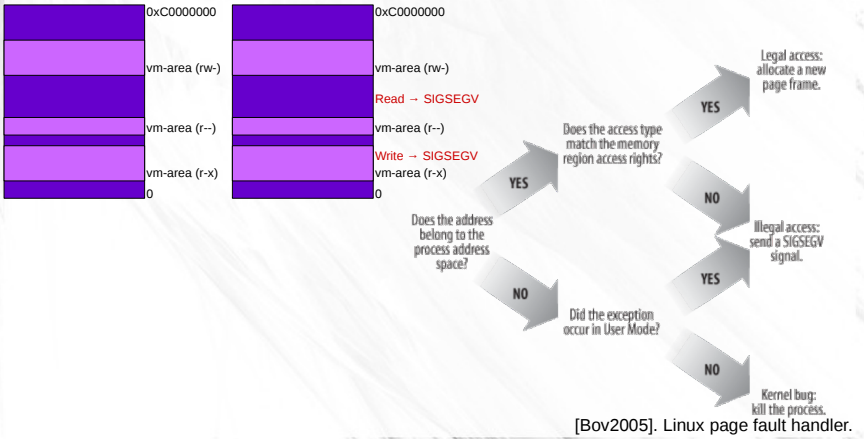
Las áreas de memoria tienen unos permisos para realizar operaciones asociadas: r,w,x.

Si un proceso referencia una dirección de memoria que no pertenece a un área de memoria válida, o accede a un área de memoria con una operación no valida, el kernel le envía el mensaje de error "Segmentation Fault". (page fault handler)

Las áreas de memoria no se solapan, una dirección válida solamente existe en una única área de memoria.

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.

El proceso mediante llamadas al kernel puede dinámicamente añadir y eliminar áreas de memoria a/de su espacio de direcciones.

Las áreas de memoria tienen unos permisos para realizar operaciones asociadas: r,w,x.

Si un proceso referencia una dirección de memoria que no pertenece a un área de memoria válida, o accede a un área de memoria con una operación no valida, el kernel le envía el mensaje de error "Segmentation Fault". (page fault handler)

Las áreas de memoria no se solapan, una dirección válida solamente existe en una única área de memoria.

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.

Un mapa de memoria de la sección de código (text section) y de la sección de variables globales inicializadas (data section) de un archivo ejecutable; un mapa de memoria de la página cero para variables globales no inicializadas (bss section) y para la pila de espacio de usuario.

* Un mapa de memoria de las secciones de texto, datos inicializados y bss para cada biblioteca compartida.
* Archivos mapeados a memoria.
* Segmentos de memoria compartida.

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 */
```


Comments

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 */
}
```


Comments

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*).

¿Cómo se asigna un descriptor de memoria?
El descriptor de memoria~(struct mm_struct) de una tarea se almacena en el campo "mm" del descriptor de proceso de la tarea (task_struct). Existen dos formas de trasvasar el descriptor de memoria a un proceso hijo: al utilizar fork() el nucleo copia (función copy_mm()) el descriptor de memoria del padre en el hijo cogiendo una estructura mm_struct de la cache correspondiente a las mm_struct (vimos el slab cache layer en la sección anterior); mediante el flag CLONE_VM al utilizar clone() el proceso hijo (hebra) comparte el descriptor de memoria del padre y se incrementa el campo mm_users de mm_struct indicando que hay un proceso más utilizando este espacio de direcciones. Las hebras Linux para el kernel son simplemente procesos que comparten ciertos recursos, incluida la memoria como hemos visto.

¿Cómo se libera un descriptor de memoria?
El núcleo llama a exit_mm() que a su vez llama a mmput() que decrementa el contador mm_users incluido en el descriptor de memoria (struct mm_struct) asociado al proceso. Si este contador llega a 0 se llama a mmdrop() para decrementar el contador de uso mm_count. Si este contador llega a valer 0 se llama a free_mm() para liberar mm_struct en la caché.

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 */
};
```


El campo **vm_mm apunta al mm_struct (descriptor de memoria) asociado al VMA.** Cada área de memoria virtual es única para el mm_struct con la que está asociada, por lo que procesos separados tienen sus VMAs propias. Sin embargo, si varias hebras comparten el espacio de direcciones, comparten todas las VMAs asociadas.

El campo **vm_start almacena la dirección inicial (más baja) del intervalo de memoria, y el campo vm_end es el primer byte tras la última dirección (más alta) del intervalo. "vm_end - vm_start" es la longitud en bytes del área de memoria cuyo intervalo es [vm_start, vm_end)**.

El campo de bits vm_flags especifica el comportamiento y proporciona información sobre las páginas contenidas en el VMA. La información tiene que ver con toda página del VMA, o al VMA como conjunto, y no a páginas individuales específicas. Algunos flag:

VM_READ Las páginas pueden ser leidas.
VM_WRITE Las páginas pueden ser escritas.
VM_EXEC Las páginas pueden ejecutarse.
VM_SHARED Las páginas son compartidas.

El flag VM_SHARED especifica si el VMA contiene una proyección que es compartida entre varios procesos. Nótese que se puede compartir un mm_struct entre varios procesos (hebras) mediante clone() con el flag CLONE_VM y un vm_area_struct en particular también se puede compartir entre procesos dependiendo del valor del flag VM_SHARED. Si el flag está activado la proyección es compartida si está desactivado la proyección es privada.

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.

Comments

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]
```


Comments

Las tres primeras líneas representan las secciones de código, datos solo lectura (constantes) y datos (variables globales) del programa vmareas. Las siguientes líneas muestran las secciones de la biblioteca de C (libc*.so). Las siguientes las secciones del enlazador dinámico (ld*.so) y la última línea la pila de ejecución.

Si una VMA es compartida o no tiene permiso de escritura el kernel mantiene solamente una copia en memoria del archivo proyectado en la VMA.

Las áreas de memoria que muestran un dispositivo 00:00 e inodo cero representan una proyección de la página cero, por lo que el área se inicializa a cero (pej. sección bss). Si la proyección no está compartida (private section), tan pronto como el proceso escribe en la región, se realiza una copia de la página y se actualiza el valor escrito en la nueva copia de la página.

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)
```


La función realiza una proyección del archivo file desde el offset con un tamaño len (proyección respaldada por archivo). Si el argumento file es NULL y el offset es 0 la proyección no será respaldada por un archivo (proyección anónima). El argumento opcional addr permite especificar la dirección inicial, a partir de la cual, buscar un intervalo libre. El argumento prot permite especificar los permisos de acceso para las páginas del VMA y el argumento flag permite especificar el resto de permisos permitidos en VMAs.

Si es posible, el intervalo se fusiona con una VMA adyacente. En otro caso, se asigna una nueva estructura vm_area_struct de la caché de bloques vm_area_cachep y se añade a la lista enlazada y al árbol rojo-negro de VMAs del descriptor de memoria (struct mm_struct) mediante la función vma_link(). La función devuelve la dirección inicial del intervalo de memoria creado.

La funcionalidad de do_mmap se exporta al espacio de usuario mediante la llamada al sistema mmap2.

void * mmap2(void *start, size_t length, int prot, int flags, int fd, off_t pgoff)

El argumento pgoff especifica el offset en páginas.

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.

La función realiza una proyección del archivo file desde el offset con un tamaño len (proyección respaldada por archivo). Si el argumento file es NULL y el offset es 0 la proyección no será respaldada por un archivo (proyección anónima). El argumento opcional addr permite especificar la dirección inicial, a partir de la cual, buscar un intervalo libre. El argumento prot permite especificar los permisos de acceso para las páginas del VMA y el argumento flag permite especificar el resto de permisos permitidos en VMAs.

Si es posible, el intervalo se fusiona con una VMA adyacente. En otro caso, se asigna una nueva estructura vm_area_struct de la caché de bloques vm_area_cachep y se añade a la lista enlazada y al árbol rojo-negro de VMAs del descriptor de memoria (struct mm_struct) mediante la función vma_link(). La función devuelve la dirección inicial del intervalo de memoria creado.

La funcionalidad de do_mmap se exporta al espacio de usuario mediante la llamada al sistema mmap2.

void * mmap2(void *start, size_t length, int prot, int flags, int fd, off_t pgoff)

El argumento pgoff especifica el offset en páginas.

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)
```


Comments
La llamada al sistema munmap() permite eliminar intervalos de direcciones desde espacio de usuario.

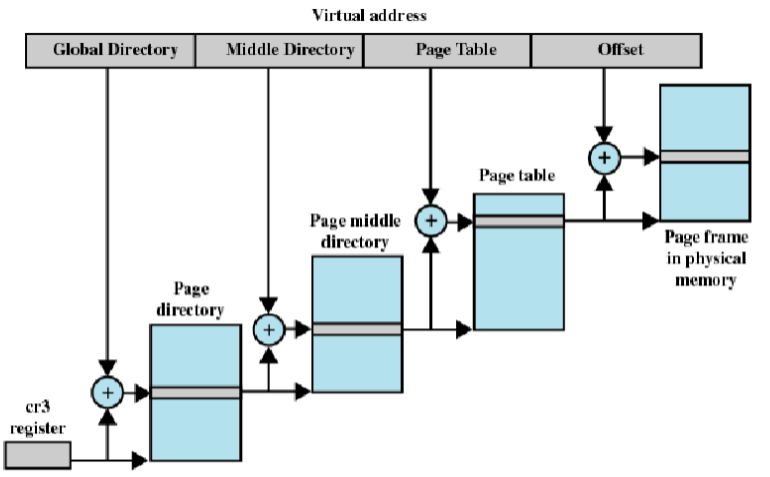
int munmap(void *start, size_t length)

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.

El campo "pgd" del descriptor de memoria (struct mm_struct) apunta al PGD del proceso. Como casi cada acceso de una página en VM debe resolverse a su dirección física correspondiente en memoria física, el rendimiento de las tablas de páginas es muy crítico. Entonces, usar TLB que es una caché HW de correspondencias virtual-a-física.

Tablas de páginas multinevel de Linux

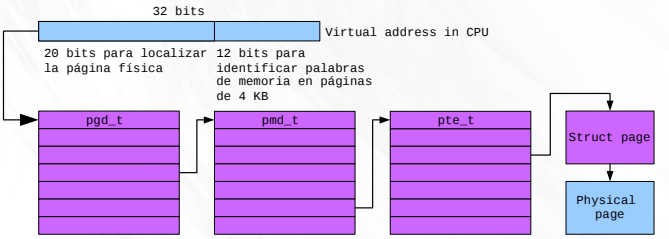


Comments

El campo "pgd" del descriptor de memoria (struct mm_struct) apunta al PGD del proceso. Como casi cada acceso de una página en VM debe resolverse a su dirección física correspondiente en memoria física, el rendimiento de las tablas de páginas es muy crítico. Entonces, usar TLB que es una caché HW de correspondencias virtual-a-física.

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.



El campo "pgd" del descriptor de memoria (struct mm_struct) apunta al PGD del proceso. Como casi cada acceso de una página en VM debe resolverse a su dirección física correspondiente en memoria física, el rendimiento de las tablas de páginas es muy crítico. Entonces, usar TLB que es una caché HW de correspondencias virtual-a-física.

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.

Justificación de por qué incluir una cache de disco en el nucleo: tiempo de acceso a disco vs tiempo acceso a memoria (milisg. vs nanosg.) y principio de localidad temporal de acceso a datos.

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).

Cada vez que el kernel efectua una operación de lectura primero comprueba si los datos que se solicitan se encuentran en la caché de páginas. Si se produce un acierto de caché (cache hit) se leen directamente de RAM, si no se encuentran en cache, fallo de caché (cache miss), el kernel tiene que planificar operaciones de E/S de bloques para leer los datos del disco.

¿qué ocurre cuando un proceso escribe en disco?

La cachés implementan principalmente dos estrategias. En la primera estrategia, una operación de escritura actualiza automáticamente la cache de memoria y el archivo en disco (write-through cache). Este enfoque tiene la ventaja de mantener coherente la cache (sincronizada y valida con respecto al almacenamiento de respaldo) sin necesidad de invalidarla.

La segunda estrategia, la que emplea Linux, se denomina **write-back** (que se refiere a la acción de escribir los datos de la caché de vuelta al almacenamiento de respaldo). En este tipo de cache los procesos realizan operaciones de escritura directamente en la caché de páginas. El almacén de respaldo no se actualiza directamente. En vez de esto, las páginas escritas en la caché se marcan como sucias (se activa el flag PG_dirty en la struct_page) y se añaden a una lista (dirty list). Periodicamente, las páginas de la lista se escriben en disco (proceso writeback), sincronizando la copia de disco con la de la caché. Tras la escritura se desactiva el flag PG_dirty de la página.

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**).

Proceso por el cual los datos son eliminados de la caché, bien para hacer sitio para entradas más relevantes de caché o para reducir la caché para ampliar la RAM disponible para otros usos.

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.

En lugar de mantener una lista LRU Linux mantiene dos: la "active list" y la "inactive list". Las páginas de la "active list" se consideran "hot" y no están disponibles para ser seleccionadas como víctimas. Las páginas de la "inactive list" están disponibles para ser seleccionadas como víctimas.

Las páginas se añaden a la active list solamente cuando son accedidas mientras todavía residen en la inactive list. Ambas listas se mantienen de forma pseudo-LRU: los elementos son añadidos al final y sacados de la cabeza, como en un TDA cola.

Ambas listas se mantienen balanceadas: si la active list llega a ser mucho más larga que la inactive list, los elementos de la cabeza de la lista activa se vuelven a colocar en la inactive list. Esta estrategia soluciona el problema de la lectura de los bloques de un archivo una sola vez.

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

Una operación de lectura de página provoca que el kernel de linux trate primeramente de encontrar los datos requeridos en la caché de páginas. Para realizar esta labor se usa el método "find_get_page()" al que se le pasan como argumentos una estructura "address_space" y un offset del archivo en páginas.

Si la página en cuestión no se encuentra en la caché, el método devuelve NULL y el kernel asigna una nueva página añadiéndola a la caché de páginas. Finalmente, los datos se leen de disco en la nueva página de la caché.

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`.

La operación de escritura tiene dos posibilidades dependiendo del objeto que representa la estructura address_space. Si representa una proyección a memoria de un archivo, simplemente se activa el flag PG_dirty del campo de flags de la struct_page que representa la página. Posteriormente el kernel escribirá esta página modificada. Si el objeto representa un archivo entonces se busca la página en la caché de páginas. Si no se encuentra se asigna una entrada y se trae el trozo de archivo correspondiente y se escribe desde espacio de usuario a un búfer del kernel y después se marcan las páginas como PG_dirty.

Como hemos visto las operaciones de escritura se realizan en la caché de páginas usando el campo PG_dirty. La escritura real a disco de páginas “sucias” ocurre en tres situaciones:

* Cuando la memoria disponible se reduce por debajo de un umbral es necesario escribir las páginas a disco para que queden "limpias" y se pueda reducir la memoria destinada a la caché.
* Cuando las páginas marcadas "dirty" superan un umbral de tiempo se escriben a disco.
* Cuando un proceso invoca las llamadas al sistema sync() o fsync() el kernel escribe las páginas marcadas "dirty".

Las flusher threads son un conjunto de hebras planificadas en pandilla que se encargan de realizar la escritura a disco de las páginas “sucias”. Primeramente, estas hebras necesitan vaciar al disco datos modificados en caché cuando la cantidad de memoria libre en el sistema cae bajo un umbral especificado (dirty_background_ratio). Con esto conseguimos liberar la memoria usada por las páginas sucias cuando queda poca memoria física. Además, de forma periódica (dirty_expire_interval) una de estas

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);
```


Como hemos visto las operaciones de escritura se realizan en la caché de páginas usando el campo PG_dirty. La escritura real a disco de páginas “sucias” ocurre en tres situaciones:

* Cuando la memoria disponible se reduce por debajo de un umbral es necesario escribir las páginas a disco para que queden "limpias" y se pueda reducir la memoria destinada a la caché.
* Cuando las páginas marcadas "dirty" superan un umbral de tiempo se escriben a disco.
* Cuando un proceso invoca las llamadas al sistema sync() o fsync() el kernel escribe las páginas marcadas "dirty".

Las flusher threads son un conjunto de hebras planificadas en pandilla que se encargan de realizar la escritura a disco de las páginas “sucias”. Primeramente, estas hebras necesitan vaciar al disco datos modificados en caché cuando la cantidad de memoria libre en el sistema cae bajo un umbral especificado (dirty_background_ratio). Con esto conseguimos liberar la memoria usada por las páginas sucias cuando queda poca memoria física. Además, de forma periódica (dirty_expire_interval) una de estas hebras es despertada por el núcleo y escribe las páginas modificadas a disco, e.d. sincroniza la caché con el disco, cumpliendo el segundo objetivo de manera que una página sucia esté demasiado tiempo sin ser escrita a disco.