TEMA 3: GESTIÓN DE MEMORIA

A. GENERALIDADES SOBRE GESTIÓN DE MEMORIA

A.1. Jerarquía de memoria

La gestión de memoria se rige por dos principios generales: a menor cantidad de memoria, acceso más rápido, a mayor cantidad de memoria, menor coste por byte.

A.2. Conceptos sobre Cachés

La memoria caché se utiliza como una copia que puede ser accedida más rápidamente que la original. El caché funciona porque los programas siguen el **principio de localidad**, es decir, los programas no acceden con la misma probabilidad a todos los datos o instrucciones, si se accede a uno, es muy probable que se acceda a los datos cercanos pronto (espacial), aunque también se puede dar la localidad temporal. Se pueden dar varios casos:

- Acierto de caché, donde el dato a acceder se encuentra en caché.
- Fallo en el caché, el dato no se encuentra en caché.
  Se puede medir el tiempo de acceso efectivo(TAE):
  TAE=probabilidad_acierto * coste_acierto + probabilidad_fallo * coste_fallo

A.3. Espacios de direcciones lógico y físico e imagen del proceso

- **Lógico:** conjunto de direcciones lógicas (relativas si se da la reubicación dinámica, o virtuales si el programa soporta memoria virtual), generadas por un programa.

- **Físico:** conjunto de direcciones físicas correspondientes a las direcciones lógicas en un instante de ejecución del programa.

- **Imagen:** el par formado por el mapa de memoria y el PCB.

A.4. Objetivos de la gestión de memoria

- **Organización:** cómo está dividida la memoria y si hay uno o varios procesos.
- **Gestión:** qué estrategias se deben seguir para obtener un rendimiento óptimo, por ejemplo
  - 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 en memoria auxiliar.
- **Protección y compartición:** el SO de los procesos de usuario y los procesos de usuario entre ellos.

Estrategias de organización de memoria

La organización de la memoria puede ser:

- *Contigua*, si la asignación de memoria para un programa se hace en un único bloque de posiciones contiguas de memoria principal, se pueden dar particiones fijas o variables.

- *No contigua*, se permite dividir el programa en bloques que se pueden colocar en zonas no contiguas, admite estrategias como la paginación, segmentación o segmentación paginada.

Las ventajas que nos puede proporcionar la paginación y la segmentación son:

- **Traducción dinámica**, todas las referencias a memoria dentro de un programa en ejecución se realizan sobre el espacio de direcciones lógico. Es decir, un programa se puede retirar y traer de vuelta y seguir ejecutándose en una nueva área de memoria.
- **No contigua**, un programa se divide en trozos (páginas o segmentos), que no tienen que estar ubicados de forma contigua.
- Todos las páginas del programa deben residir en memoria principal durante la ejecución del programa.

Por otra parte, también se utiliza una técnica llamada **swapping**, o intercambio, que consiste en intercambiar procesos entre memoria principal y auxiliar.
El proceso pasa al estado SUSPENDIDO_BLOQUEADO, y el espacio ocupado pasa a disco.
Esta técnica tiene una serie de características:

- El almacenamiento auxiliar debe ser un disco rápido.
- El factor principal de tiempo es el intercambio MA<->MP.
- El swapper se encarga de: seleccionar procesos para retirarlos de MP, seleccionar procesos para incorporarlos a MP y gestionar y asignar el espacio de intercambio.

B. ORGANIZACIÓN DE LA MEMORIA VIRTUAL

B.1. Concepto

La memoria virtual surge de:

- La necesidad de paginación y segmentación básica.
- Cuando el tamaño del programa (código, datos, pila y regiones), excede la cantidad de memoria física disponible.
- El número de procesos en MP aumenta (grado de multiprogramación).

Resuelve el problema del crecimiento dinámico del mapa de memoria de los procesos, y para llevarlo a cabo se requiere usar dos niveles de la jerarquía de memoria para almacenar el

programa: **memoria principal** (residen las partes del programa necesarias en un momento dado, o *conjunto residente*) y **memoria auxiliar** (reside el espacio de direcciones completo del programa).

Los requisitos para su implementación son:

- Disponer de información sobre qué partes del programa se encuentran en MP y en MA, en la **tabla de ubicación en disco, TUD**.
- Política de resolución de un acceso a memoria situado en una parte que no reside en MP.
- Política de movimiento de partes del espacio de direcciones entre MP y MA.

B.2. Unidad de gestión de memoria

Se utiliza el MMU, un dispositivo gestionado por el SO, que traduce direcciones virtuales a direcciones físicas. Se encarga de sumar el valor del registro base a cada dirección generada por la ejecución del programa, siendo éste resultado el que se usa para acceder en el bus de memoria a la dirección física deseada.

El MMU, tiene unos registros TLB (Transation Look-aside Buffer) que mantienen la correspondencia entre página virtual/física resueltas con anterioridad.

Las responsabilidades del MMU son:

- Si TLB hit, realizar la traducción.
- Si TLB miss, usar la *Tabla de Páginas* para traducir, teniendo en cuenta que si la parte del espacio de direcciones que contiene la dirección resultado de la traducción reside en MP, carga el nuevo valor, si no genera una `page fault exception`.

B.3. Memoria virtual paginada

La organización del espacio no es continua, se asigna la memoria mediante bloques de tamaño fijo (**marcos de página**), cuyo tamaño va de 0.5 a 8 KB.
Las direcciones se interpretan a dos niveles:

- Los bits más significativos = página virtual donde está la dirección.
- Bits menos significativos = offset en marco de página.

La correspondencia página virtual y marco de página se almacena en la *Tabla de páginas*, donde se introduce la dirección física del marco de página.

- **Dirección virtual:** generada por la CPU, contiene un par:
  - El número de página, que determina la entrada en la TP (bits + significativos).
  - Offset, completa la dirección física (bits menos significativos).

- **Dirección física:** es la real en memoria principal, se calcula con la *dirección base del marco de página* (almacenada en TP) y el *offset* (se suma).

Estructuras de MVP:

Se utilizan tres estructuras principales:

- **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.

Los vamos viendo por orden:

Tabla de páginas:

Contiene una entrada por cada página virtual del proceso que a su vez contiene:

- Dirección base del marco.
- Protección o modo de acceso a la página.
- Bit de validez.
- Bit de modificación.

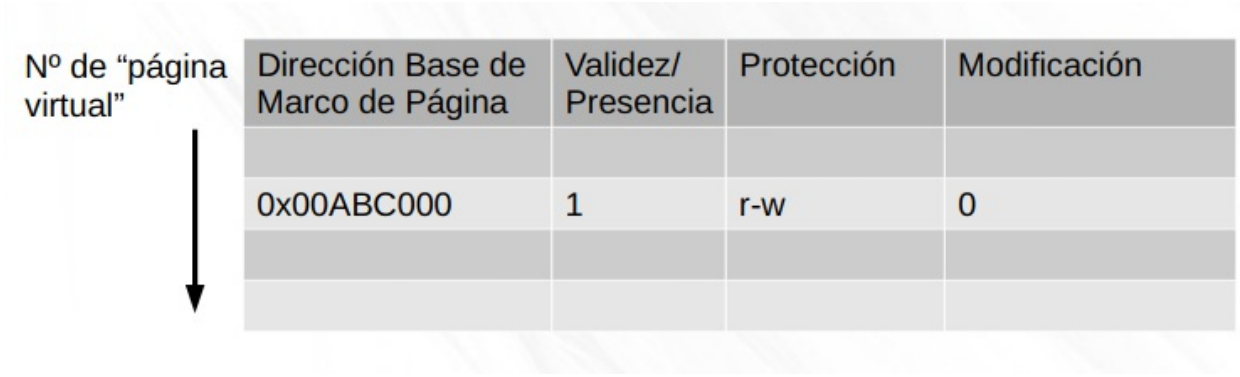


Tabla de Ubicación en Disco:

Contiene una entrada por cada página virtual con:

- Identificación del dispositivo lógico que tiene la MA.
- Identificación del bloque que contiene la copia de la página.
  Además, crece hacia abajo.

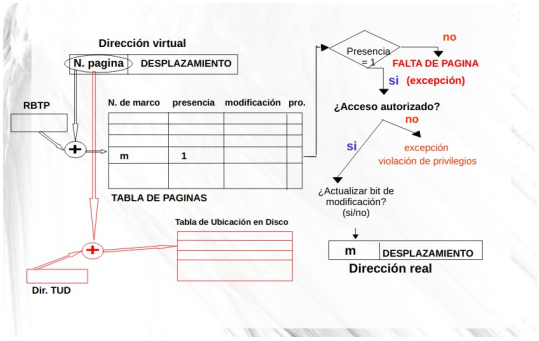
B.4. Memoria virtual mediante paginación por demanda

Se basa en el **modelo de localidad**, donde una localidad es un conjunto de páginas que se

utilizan durante un periodo de tiempo, que durante la ejecución de un programa va variando.

Las características de esta estructura son:

- Los programas residen en un dispositivo de intercambio, o *backing store*.
- Al crear el proceso, el SO sólo carga en memoria RAM un subconjunto de páginas.
- La tabla de páginas se inicializa con los valores correctos, páginas válidas y cargadas en RAM; páginas válidas pero no cargadas y páginas no válidas.



- Después de inicializar, se cambia el proceso a LISTO.

Errores posibles: **PAGE FAULT**

Gestión del error:

```
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.
```

En Linux se distingue entre errores de programación relativos a acceso a memoria y errores debidos a falta de página.

Implementación de la Tabla de Páginas

- 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.

Problema de tamaño de página

La tabla de páginas ocupa demasiado tamaño, por eso se usa la *programación multinivel*.

B.5. Programación multinivel

Se soluciona el problema de tamaño de página al paginar las tablas de páginas, es decir, se divide 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. ?¿?¿?

No se hace explícita la paginación a nivel de página hasta que se haga válida esa parte del espacio de direcciones, las partes del espacio de direcciones virtual que no son válidas no tienen tabla de páginas.

Paginación a dos niveles

La dirección virtual se interpreta: ver ejemplo.

Compartición de páginas

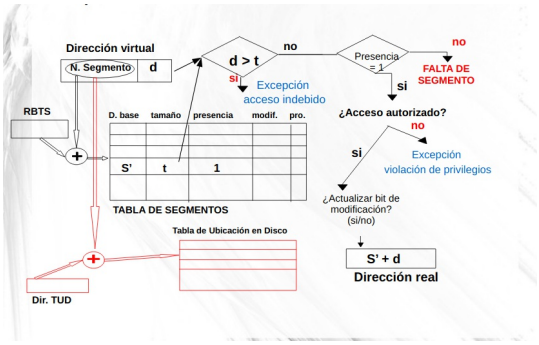
Una página que contenga código puede ser compartida, en ese caso, las TP de los procesos que comparten reflejan la misma dirección base de marco.
Las páginas compartidas no se sustituyen al reemplazar páginas (ejemplo: libc o ld.so).

B.6. Memoria virtual Segmentada

- **Dirección virtual** tupla formada por (id_segmento, offset), que se genera en la CPU y se mantiene en:
    - Registro de segmento, contiene el id de segmento utilizado en el momento.

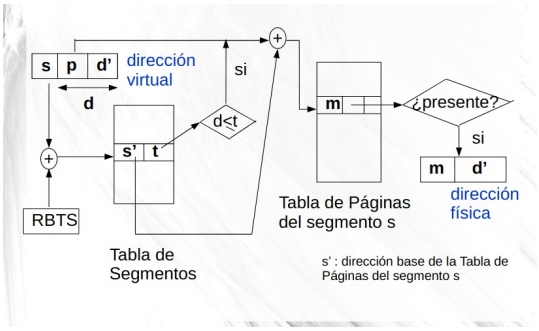
- Registro de offset, desplazamiento en el segmento actual (identifica la dirección virtual del segmento).

- **Dirección física**, dirección real que se calcula con: dirección de memoria principal donde empieza el segmento virtual (en TS), offset, sumando.



B.7. Segmentación paginada

Problemas de segmentación: variabilidad de tamaño de segmentos y requisito de memoria contigua, complica la gestión.
Problemas de paginación: complica compartición y protección.



C. GESTIÓN DE LA MEMORIA VIRTUAL

C.1. Conceptos

En la gestión de memoria virtual paginada, se utilizan **criterios de clasificación respecto** a:

- Políticas de asignación de memoria principal (fija o variable)
- Políticas de búsqueda de páginas en MP (paginación por demanda o anticipada)
- Políticas de sustitución de páginas en MP (global o local).

Independientemente de la política que se utilice, *siempre se deben cumplir los siguientes criterios*:

- Páginas limpias frente a sucias (minimizar transferencias MP-SWAP).
- Las últimas seleccionadas son las compartidas.
- Páginas especiales, puede haber algún marco de página bloqueado (búferes de E/S mientras se realiza transferencia fifo).

Además, *el tamaño de página sí importa*:

- Menor tamaño
  - Aumento de tamaño de tablas de página.
  - Aumento de transferencias MP-swap.

    - Reducen fragmentación interna.
- Mayor tamaño:
    - Mucha información no usada está en MP
    - Más fragmentación interna
      Lo adecuado es buscar un equilibrio

C.2. Algoritmos de sustitución:

Se pueden tener varias combinaciones en cuanto a tipo de asignación de memoria principal y tipo de sustitución de página:

- Asignación fija y sustitución local.
- Asignación variable y sustitución global.
- Asignación variable y sustitución global.

Los distintos algoritmos de sustitución se pueden clasificar en:

- Óptimo, sustituye la página que no se va a referenciar en un futuro, o que lo hará más tarde.
- FIFO, sustituye la más antigua.
- LRU, sustituye la que se referenció hace más tiempo.
- Algoritmo de reloj, lo vemos ahora.

Algoritmo del reloj

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 son una lista circular y un puntero a la página visitada hace más tiempo.
Para seleccionar una página se:

```
1. Consultar marco actual
2. R=0?
  si: selecciona para sustituir e incrementa posicion
  no: ir al siguiente marco y volver al paso 1
```

Aún así, la cantidad de memoria principal disponible influye más en las faltas de página que el algoritmo de sustitución utilizado.

C.3. Hiperpaginación (trashing)

Un proceso está hiperpaginado si está más tiempo haciendo E/S sobre backing store que ejecutándose, es decir, que la tasa de faltas de página es alta.
Si un sistema tiene muchos procesos hiperaginados puede causar:

- Poco uso de CPU porque falta página (E/S).
- Poco uso de CPU (crea nuevos para aumentar uso)
- Los nuevos entran en trashing

En resumen, el SO resuelve faltas de página, entonces el tiempo de núcleo aumenta y el tiempo útil de computación cae.

Soluciones:

- Asegurar que cada proceso tenga asignado un espacio acorde con su comportamiento (asignación variable de marcos), es decir, se tiene que estimar qué conjunto residente de páginas es el óptimo.
- Actuar sobre el grado de multiprogramación (regulación de carga).

Comportamiento de los programas

El **comportamiento de ejecución** es la secuencia de referencias a página que realiza un proceso, es importante para maximizar el rendimiento del sistema de memoria virtual.

C.4 Principio de localidad

Los programas referencian una pequeña parte del espacio de direcciones durante un determinado tiempo. Puede ser espacial y temporal:

- **Temporal**, una posición refereciada recientemente tiene que una alta probabilidad de ser referenciada próximamente. Provocada por ciclos o contadores en ciclos.
- **Espacial**, si una posición ha sido referenciada recientemente, hay una alta probabilidad de que las posiciones adyacentes sean referenciadas. Provocada por código en secuencia y arrays.

MODELO DE CONJUNTO DE TRABAJO

El **conjunto de trabajo de páginas** (working set), es el conjunto de páginas referenciado por el proceso durante el intervalo de tiempo (t-μ, t).
Mientras el conjunto de trabajo de páginas pueda estar en MP, el nº de faltas es bajo. Pero si se eliminan de MP páginas del conjunto de trabajo, aumentan las faltas.

Propiedades del conjunto:

- Los conjuntos son transitorios, difieren en su composición sustancialmente.
- No se puede predecir el tamaño ni composición de un conjunto de trabajo futuro.

Requisitos del conjunto:

- Un proceso solo se puede ejecutar si su conjunto está en memoria principal.
- Una página no se puede retirar de memoria principal si forma parte del conjunto actual.

El modelo presenta una estrategia:

- Si el número de páginas referenciadas aumenta y no hay espacio en memoria para ubicarlas, el proceso se intercambia a disco.
- Al sacar de memoria varios procesos, los demás acaban antes, incluso los que se sacan de memoria.

¿Cómo funciona el algoritmo del conjunto de trabajo?

En cada referencia, determina el conjunto de trabajo (t-μ, t), y solo se mantienen esas páginas en MP.

ALGORITMO DE FRECUENCIA DE FALTA DE PÁGINA

Consiste en que para ajustar el conjunto de páginas del proceso que residen en memoria principal (conjunto residente), se usan los intervalos de tiempo entre dos faltas de páginas consecutivas:

- Si el intervalo es grande (mayor que un umbral), todas las páginas no referenciadas en dicho intervalo son retiradas de MP.

- Si no, la nueva página se incluye en el conjunto residente.

Este algoritmo garantiza que el conjunto residente crece cuando las faltas son frecuentes, y decrece cuando no lo son.

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

```
tc= instante de tº de la falta actual de página
t(c-1)=instante de la falta anterior
Z=conjunto de páginas referenciadas
R= conjunto de páginas residentes
```

D. GESTIÓN DE MEMORIA EN LINUX

D.1. Gestión de memoria a bajo nivel

El kernel gestiona el uso de la memoria física, y junto con el hardware trabajan con páginas, cuyo tamaño depende de la arquitectura.

Cada marco de página es representada por `struct page`:

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

RESTRICCIONES

Una página puede ser utilizada por:

- La caché de páginas. El campo mapping apunta al objeto representado por `struct adres_space`.
- 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.

Cualquier página no puede utilizarse para cualquier tarea (restricciones del HW), por tanto, se divide la memoria física en zonas de memoria. (foto: x86)

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

Restricciones

Además, se usa un tipo, `gfp_t`, que especifica el tipo de memoria que se solicita, mediante tres categorías de flags:

- Modificadores de acción (GPF_WAIT, puede entrar en sleep, GPF_IO)
- Modificadores de zona(GFP_DMA).
- Tipos, más abstracto, por ejemplo, GFP_KERNEL (solicitud para kernel), ó GFP_USER (solicitar para el espacio de usuario de un proceso).

API

Son interfaces para la asignación de memoria física que proporcionan memoria en múltiplos de páginas físicas.

- Asigna 2^order páginas físicas contiguas y devuelve un puntero a la struct page de la primera, si falla, NULL.
  `struct page* alloc_pages(gfp_t gfp_mask, unsigned int order)`

- Asigna 2^order páginas físicas contiguas y devuelve la dirección lógica de la primera.
  `unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)`

- Proporcionan/liberan memoria en chunks de bytes
  `void * kmalloc(size_t size, gfp_t flags)`
  `void kfree(const void *ptr)`

CACHÉ DE BLOQUES

La asignación y liberación de estructuras de datos es muy común en el kernel del SO. Para agilizar esta solicitud Linux utiliza el **nivel de bloques**.
El nivel de bloques actúa como una caché de estructuras genérica:

- Existe una caché para cada tipo de estructura distinta `struct task_struct cache` , `struct inode cache` .
- Cada cache tiene varios bloques de una o más páginas físicas contiguas.
- Cada bloque tiene estructuras de su tipo

Funcionamiento:

Cada bloque puede estar: lleno, parcialmente lleno o vacío.
El procedimiento cuando el kernel solicita una nueva estructura es:

- La solicitud se satisface en un bloque parcialmente lleno.
- Si no hay, se llena uno vacío.
- Si no hay, se crea uno nuevo.

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

D.2. ESPACIO DE DIRECCIONES DEL PROCESO

Linux utiliza MV, a cada proceso se le asigna un espacio de memoria plano único (32 o 64bits), aunque se puede compartir el espacio de memoria (CLONE_VM).
Parte de este espacio solo es accesible en modo kernel, el proceso solo tiene permiso para acceder a determinados intervalos de direcciones de memoria (vm-areas).

Además, Linux distingue entre los errores de programación por acceso a memoria y errores por falta de página.

VM-ÁREAS:

Contenido:

- 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 0, 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:

Representa 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*/

  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 */
  }
```

Para asignar el descriptor de memoria:

- Se copia el descriptor `fork()`.
- Se comparte el descriptor de memoria mediante el flag CLONE_VM de la llamada `clone()`.

Para liberar el descriptor:

- El núcleo decrementa `mm_users`. Si llega a 0 se decrementa `mm_count`. Si éste último llega a 0 se libera de la caché (slab cache).

Área de memoria:

Describe un espacio 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 */ }
```

Creación y expansión:

Para poder crear y expandir Vm-áreas usamos `do_mmap()`, que nos permite:

- Expandir un VMA ya existente (el intervalo añadido es adyacente con mismos permisos).
- Crear una nueva VMA para representar 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)`

`do_mmap()` crea una proyección del archivo file a partir del *offset* con un tamaño de *len* bytes:

- Si file=NULL y offset=0, es 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` especifica permisos de acceso.
- `flag`, especifica el resto de permisos

Eliminación

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

Usamos `do_munmap()`, donde:

- `mm` especifica el descriptor de memoria del que se va a eliminar el intervalo de memoria que comienza en start y tiene len bytes.

D.3. TABLAS DE PÁGINA MULTINIVEL EN LINUX

Las direcciones virtuales se transforman en físicas mediante tablas de páginas (3 niveles):

- **directorio global de páginas(PGD)**, con un array de `pgd_t`.
  - Las entradas apuntan a entradas de la tabla de segundo nivel.
- **page middle directory (PMD)**, un array de `pmd_t`.
  - - Las entradas apuntan a entradas de la tabla de primer nivel.
- **tabla de páginas**, entradas de la tabla de páginas de tipo `pte_t`, que apuntan a `struct_page`.

Un enfoque que mejora esta alternativa es usar TLB en la MMU.

D.4. CACHÉ DE PÁGINAS

Cuando el kernel quiere leer algo de disco, primero comprueba si los datos están en el caché de páginas:

- Si hit, lee de caché.
- Si miss, solicita E/S a disco.

Cuando el kernel quiere escribir algo en disco puede:

- Write through cache, actualizar memoria y disco.
- Write back cache, escribir en la cache de páginas.

Cache eviction

Ocurre cuando se eliminan datos de la caché junto con la estrategia de qué datos eliminar. En Linux:

- Selecciona páginas limpias (no `PG_dirty`) y las reemplaza.
- Si no existen suficientes limpias en cache, se fuerza un proceso de escritura a disco para hacer disponibles más páginas limpias.
- Se toma la decisión de qué páginas limpias eliminar.

Selección de víctima

- *LRU*, Least recently used, requiere mantener información de cuando se accede a cada página y seleccionar las páginas con el tiempo de acceso más antiguo. Problema: único

acceso. Solución: *active e inactive list*

- Las páginas del active no se pueden escoger como víctimas.
- Solo se añaden a la active list si son accedidas mientras residen en la inactive list.
- Las de inactive se seleccionan.

Lectura

Se usa un objeto para gestionar entradas de la caché y operaciones de E/S de páginas: `struct address_space` (representa páginas físicas de un archivo)

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

Para leer primero se busca la información en la caché de páginas:

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

Si devuelve NULL, se asigna una nueva página y se añade a la caché de páginas, entonces se convierte en una operación de lectura de disco.

Escritura

Para la operación de escritura existen dos posibilidades, dependiendo del objeto que representa `struct address_space`:

- Si es una proyección a memoria se activa `PG_DIRTY` de la struct_page.
- Si es un archivo:
  - Se busca en el caché de páginas.
  - Si no se encuentra se asigna una entrada se devuelve el trozo de archivo correspondiente a una página.
  - Se escribe la información en la página y se activa `PG_DIRTY`.

Flusher threads

La escritura real a disco de páginas sucias ocurre cuando:

- La memoria disponible cae por debajo de un umbral.
- Las páginas sucias superan un umbral de tiempo.
- Cuando un proceso invoca a la llamada `sync()` o `fsync()`.

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

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