3.1. Gestión de memoria principal

Jerarquía de memoria

Dos principios sobre memoria:

- Menor capacidad, acceso más rápido
- Mayor capacidad, 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

La **caché** es la memoria de acceso rápido de un ordenador, que guarda temporalmente los datos recientemente procesados. Es una copia que puede ser accedida más rápidamente que el original. La idea de este tipo de memoria es hacer los casos frecuentes eficientes, los caminos infrecuentes no importan tanto. Decimos que hay un **acierto de caché** si el item buscado está en la caché. Denominamos **fallo de caché** a que el item no esté en cachéy haya que realizar la operación completa (recuperar el bloque en el que se encuentra en la memoria principal y traerlo a la caché). $$ Tiempo de Acceso Efectivo (TAE) = Probabilidad*{acierto} \cdot coste*{acierto} + Probabilidad*{fallo} \cdot coste*{fallo} $$

Funciona porque los programas no son aleatorios, explotan la localidad (principio de localidad, sólo unas pocas porciones del proceso se necesitarán a lo largo de un periodo de tiempo corto).

Espacio de direcciones lógico y espacio de direcciones físico

Espacio de direcciones lógico: conjunto de conjunto de direcciones lógicas o virtuales generadas por un programa. Direcciones lógicas son aquellas a las que hacen referencia las instrucciones de un programa.

Espacio de direcciones físico: conjunto de direcciones físicas correspondientes a las direcciones lógicas en un instante dado.

El **mapa de memoria** de un proceso es una estructura de datos que contiene, entre otras cosas, información sobre las relaciones entre direcciones lógicas y físicas.

La imagen de un proceso está formada por el mapa de memoria y el PCB.

3.2. Memoria virtual: Organización

Tipos de organización de memoria

Podemos clasificar las organizaciones de memoria en dos tipos:

- **Contiguas**: la asignación de almacenamiento para un programa se hace en un único bloque de posiciones coniguas de memoria (particiones fijas y particiones variables).
- **No contiguas**: permiten dividir el programa (espacio de direcciones del proceso) en bloques o segmentos que se pueden colocar en zonas

no necesariamente continuas de memoria principal (paginación, segmentación y segmentación paginada). Este tipo de organización es el usado en la actualidad.

Intercambio (Swapping)

El **intercambio** o **swapping** consiste en mover un proceso o parte de él entre memoria y un almacenamiento auxiliar. Este almacenamiento auxiliar debe ser un disco rápido con espacio para albergar las imágenes de memoria de los procesos de usuario. El factor principal en el tiempo de intercambio es el tiempo de transferencia. El **intercambiador** tiene las siguientes responsabilidades:

- Seleccionar procesos para retirarlos de memoria principal
- Seleccionar procesos para incorporarlos a memoria principal
- Gestionar y asignar el espacio de intercambio

Concepto de memoria virtual

El tamaño del programa, los datos y la pila pueden exceder la cantidad de memoria física disponible para él. Por ello se usa un almacenamiento a dos niveles:

- Memoria principal: partes del proceso necesarias en un momento dado (**memoria real**)
- Memoria secundaria: espacio de direcciones completo del proceso (**memoria virtual**)

La memoria virtual resuelve el problema del crecimiento dinámico de los procesos y permite aumentar el grado de multiprogramación. Para utilizarla,

es necesario: saber qué se encuentra en memoria principal y una política de movimiento entre memoria principal y secundaria.

La memoria virtual soluciona el poco espacio de memoria principal frente a la gran cantidad de procesos a ejecutar, pero no optimiza la velocidad de ejecución.

Unidad de Gestión de Memoria

La **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 añade a cada dirección generada por el proceso de usuario al mismo tiempo que es enviado a memoria.

El programa de usuario trata solo con direcciones lógicas, nunca con direcciones reales.

Además de la traducción, el MMU deberá detectar si la dirección aludida se encuentra o no en memoria principal y generar una excepción si no se encuentra. La MMU cuenta con sus propios registros. Ejemplo de MMU: El MIPS R2000/3000.

Paginación

El espacio de direcciones físicas de un proceso puede ser no contiguo.

La memoria física se divide en bloques de tamaño fijo, denominados **marcos de página**. El tamaño es potencia de dos, de 0.5 a 8 Kb.

El espacio lógico de un proceso se divide en bloques del mismo tamaño, denominados **páginas**.

Los marcos de páginas contendrán páginas de los procesos.

Las **direcciones lógicas**, que son las que genera la CPU se dividen en *número de página* y *desplazamiento* dentro de la página.

Las **direcciones físicas** se dividen en *número de marco* (dirección base del marco donde está almacenada la página) y desplazamiento.

La paginación se basa en tener en memoria principal las páginas que serán necesarias para el proceso. Para esto es importante tener un control sobre las páginas que están ya cargadas, y para llevarlo a cabo se usa una tabla de páginas, que recogerá información sobre cada página del proceso, si se encuentra cargada o no o si ha sido modificada.

A veces los procesos son muy grandes y tienen tablas de páginas muy grandes y tenerlas cargadas en memoria sería un gasto considerable de espacio, por lo que las mismas tablas de páginas se ven sujetas a la memoria virtual. Para esto se usan esquemas de paginación de varios niveles, donde el número de páginas totales se divide entre el tamaño de una página para crear una tabla de primer nivel, en cada entrada de esta se apunta a una tabla de segundo nivel que guardará la información sobre las páginas del proceso.

Tabla de páginas

Cuando la CPU genere una dirección lógica será necesario traducirla a la dirección física correspondiente, la **tabla de páginas** mantiene información necesaria para realizar dicha traducción. Existe una tabla de páginas por proceso.

La **tabla de ubicación en disco** (única para cada proceso) contiene la ubicación de cada página en el almacenamiento secundario.

La **tabla de marcos de página** (usada por el SO) contiene información sobre cada marco de página.

Contenido de la tabla de páginas

Esta tabla posee una entrada por cada página del proceso:

- **Número de marco**: dirección base del marco en el que está almacenada la página si está en memoria principal
- **Bit de presencia**: 1 si la página está cargada en memoria principal, 0 si no está cargada o no es válida
- **Bit de modificación**: 1 si se ha modificado la página desde que está en memoria principal, en cuyo caso habrá que escribir la página cuando sea sustituida
- **Bits de protección**: indican que permisos tiene el proceso sobre la página

La tabla de páginas se mantiene en memoria principal. El **registro base de la tabla de páginas** (RBTP) apunta a la tabla de páginas (suele almacenarse en el PCB del proceso). Tiene dos problemas principales: cada acceso a una instrucción requiere dos accesos a memoria (resuelto con TLB) y el tamaño de la tabla de páginas (resuelto con multipaginación). Ambas soluciones se detallan a continuación.

Buffer de traducción adelantada

Cada referencia a memoria virtual puede generar dos accesos a memoria (uno para obtener la entrada de la tabla de páginas y otro para obtener el dato deseado). Para evitar que se doble el tiempo de acceso a memoria se usa el **buffer de traducción adelantada, TLB** (Translation Lookaside

Buffer), una caché especial que contiene aquellas entradas de la tabla de páginas usadas hace menos tiempo.

Dada una dirección lógica, el procesador examina en primer lugar la TLB. Si la entrada de la tabla de páginas buscada está presente (acierto en la TLB), se obtiene el número de marco y se forma la dirección real. Si no se encuentra la entrada de la tabla de páginas (fallo en la TLB), el procesador emplea el número de página como índice para buscar en la tabla de páginas del proceso su entrada correspondiente. Si se encuentra activo el bit de presencia, la página está en memoria principal y el procesador puede obtener el número de marco en que se encuentra para formar la dirección física. El procesador, además, actualiza la TLB para incluir esta nueva entrada de la tabla de páginas. Si el bit de presencia no está activo, la página buscada no está en memoria principal, se produce un fallo en el acceso a la memoria que denominamos **falta de página**.

Falta de página

Cuando se tiene una falta de página ocurre lo siguiente:

1. Bloquear proceso
2. Encontrar la ubicación en disco de la página solicitada (tabla de ubicación en disco)
3. Encontrar un marco libre. Si hay se elige y se pasa al 4. Si no hay, tenemos dos opciones: mantener bloqueado el proceso hasta que se quede uno libre o desplazar una página de memoria principal
4. Cargar la página desde disco al marco de memoria principal
5. Actualizar tablas (bit de presencia = 1, número de marco,...)
6. Desbloquear proceso
7. Reiniciar la instrucción que provocó la falta de página

\newpage

Tamaño de página

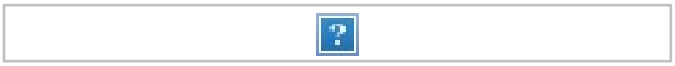
Si tenemos, por ejemplo, unas direcciones virtuales de 32 bits y un tamaño de página de 4KB (2^{12}B). El tamaño del campo de desplazamiento serán 12 bits y el tamaño del número de página virtual 20 bits. Entonces tendremos 2^{20} páginas virtuales. No podemos dedicar tanta memoria a la tabla de páginas así que las tablas de página se someten también a paginación en un proceso conocido como **paginación multinivel**.

Paginación multinivel

Cuando las tablas de página están sujetas a paginación tenemos multipaginación. La partición de la tabla de páginas permite al SO dejar particiones no usadas sin cargar hasta que el proceso las necesite. Aquellas porciones del espacio de direcciones que no se usan no necesitan tener su parte de la tabla de páginas cargada en memoria.

Dividimos la tabla de páginas en partes del tamaño de una página. La dirección lógica se divide en:

Número de página (k bits) p1	Desplazamiento de página (n-k bits) p2	Desplazamiento de página (m bits) d



Páginas compartidas

Si varios procesos comparten las mismas páginas y no las modifican, podemos tener una única copia del código en memoria.

Segmentación

Esquema de organización de memoria que permite al usuario contemplar la memoria como si constara de varios espacios de direcciones o segmentos, un programa se verá como una colección de unidades lógicas, llamadas **segmentos** (funciones, pila, tabla de símbolos, matrices, ...).

Tabla de segmentos

Una dirección lógica es una tupla *(número de segmento, desplazamiento)*

La tabla de segmentos aplica direcciones bidimensionales definidas por el usuario en direcciones físicas de una dimensión. Cada entrada de la tabla tiene los siguientes elementos:

- **Base**: dirección física donde reside el inicio del segmento en memoria
- **Tamaño**: longitud del segmento
- **Bit de presencia**
- **Bit de modificación**
- **Bits de protección**

La tabla de segmentos se mantiene en memoria principal. Cuando un proceso se está ejecutando, el **Registro Base de la Tabla de Segmentos** (RBTS) apunta a la tabla de segmentos (suele almacenarse en el PCB del proceso). El **Registro Longitud de la Tabla de Segmentos** (STLR) indica el número de segmentos del proceso; el número de segmento *s* generado en una dirección lógica es legal si $s < STLR$. El número de segmento de la dirección virtual se emplea como índice de la tabla para buscar la dirección de la memoria principal correspondiente al comienzo del segmento. Esta se añade a la parte de desplazamiento de la dirección virtual para generar la dirección física deseada.

Ventajas

- Simplifica el tratamiento de estructuras que pueden crecer
- Permiten que los programas se modifiquen de forma independiente sin requerir del programa entero
- Da soporte a la compartición entre procesos
- Mecanismos de protección

Segmentación paginada

Tanto paginación como segmentación tienen sus propias ventajas y desventajas. La variabilidad del tamaño de los segmentos y el requisito de memoria contigua dentro de un segmento complican la gestión de memoria principal y secundaria. Por otro lado, la paginación simplifica la gestión pero complica los temas de compartición y protección. La **segmentación paginada** combina ambos enfoques, obteniendo la mayoría de las ventajas de la segmentación y eliminando los problemas de una gestión de memoria compleja.

Cuando un proceso está en ejecución, un registro mantiene la dirección de comienzo de la tabla de segmentos de dicho proceso. A partir de la dirección virtual, el procesador utiliza la parte correspondiente al número de segmento para indexar dentro de la tabla de segmentos del proceso para encontrar la tabla de páginas de dicho segmento. Después, la parte correspondiente al número de página de la dirección virtual original se utiliza para indexar la tabla de páginas y buscar el correspondiente número de marco. Este se combina con el desplazamiento correspondiente de la dirección virtual para generar la dirección física requerida.

3.3. Memoria vitual: Gestión

Nos centraremos en la gestión de memoria virtual con paginación. Puede no ser posible traer todas las páginas de un programa a memoria principal para preparar su ejecución. Criterios de clasificación:

- **Políticas de asignación**:
    - *Fija*: número fijo de páginas que ejecutar
    - *Variable*: número de marcos variable en tiempo de ejecución
- **Políticas de búsqueda (recuperación)**: determina cuándo una página se trae a memoria principal
    - *Paginación por demanda*: una página se trae a memoria principal cuando el proceso incurre en una falta de página
    - *Paginación anticipada*: el SO trae a memoria páginas que no se han demandado o cuando comienza la ejecuión o bien cuando se produce una falta de página**(!=prepaginación)**
    - **Políticas de sustitución (reemplazo)**: se aplican cuando no hay marcos libres y es necesario traer una página a memoria principal para solucionar una falta de página
        - *Sustitución local*: elige páginas en memoria principal del proceso que ha producido la falta de página
        - *Sustitución global*: considera todas las páginas de la memoria como candidatas para reemplazar

Independientemente de la política de sustitución utilizada, existen ciertos criterios que siempre deben cumplirse:

- Páginas "limpias" (no han sido modificadas) frente a "sucias" (modificadas, es necesario copiarlas en memoria secundaria al eliminarlas de la principal): se pretende miniminzar el coste de

transferencia. De esta forma, el tiempo en realizar la asignación es menor, ya que no hay que hacer copia.
- Páginas compartidas: una página usada por muchos procesos se mantendrá en memoria principal, para reducir el número de faltas de página
- Páginas especiales: algunos marcos pueden estar bloqueados, no deben ser eliminados de memoria hasta que ocurra algo (ej: bufferes de E/S mientras se realiza una transferencia)

Influencia del tamaño de página

- Cuanto más pequeñas

$\shortuparrow$ tamaño de las tablas de página

$\shortuparrow$ número de transferencias memoria principal $\rightarrow$ disco

$\shortdownarrow$ fragmentación interna

- Cuanto más grandes

$\shortuparrow$ información que no será usada ocupando memoria principal

$\shortuparrow$ fragmentación interna

- Búsqueda de equilibrio

Algoritmos de sustitución

Podemos tener las siguientes combinaciones:

- **Asignación fija y sustitución local**: se tiene un proceso que se ejecuta en la memoria principal con un número fijo de páginas previamente decidido. Cuando se produce una falta de página el SO elige la página a reemplazar entre las de dicho proceso.
- **Asignación variable y sustitución local**: se tienen varios procesos con un número de marcos fijos que se va evaluando de vez en cuando. Cuando se produce una falta de página, se selecciona la página a reemplazar de entre las del conjunto residente[^1] del proceso que sufre la falta de página.
- **Asignación variable y sustitución global**: se tienen varios procesos en memoria principal, cada uno de ellos con cierto número de marcos asignados. Cuando se produce una falta de página, se añade un marco libre al conjunto residente del proceso y se carga la página. Cuando no hay marco libres, la selección de la página a eliminar se realiza entre todos los marcos de la memoria.

[^1]: El conjunto residente es el número de marcos asignado a un proceso

Veremos distintos algoritmos generados cuando no hay suficiente espacio en memoria:

Óptimo

Sustituye la página que no se va a referenciar en un futuro o la que se referencia más tarde. Es imposible de implementar, porque requiere que el SO tenga un perfecto conocimiento de los eventos futuros. Sin embargo, se utiliza como un estándar a partir del cual contrastar algoritmos reales.

FIFO

Trata los marcos de página ocupados como si se tratase de un buffer circular, sustituye de forma cíclica la página más antigua cargada en

memoria principal. Es la política de reemplazo más sencilla de implementar. El razonamiento tras este modelo es que una página traída a memoria hace mucho tiempo puede haber dejado de utilizarse (razonamiento a menudo erróneo).

LRU

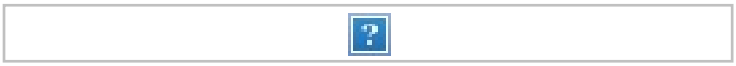
Política de reemplazo de la página usada menos recientemente (Least Recently Used). Sustituye la página que fue objeto de la referencia más antigua, debido al principio de proximidad, esta página es la que tiene menor probabilidad de ser referenciada en un futuro próximo. El problema de esta alternativa es la dificultad en su implementación, hay diferentes opciones muy costosas: marcar cada página con el instante de su última referencia, manejarlo mediante hardware, ...

Algoritmo del reloj

Es una variante de LRU menos costosa. Cada página tiene asociado un bit de referencia R (activado por el hardware). 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. ¿R == 0?
    - No. R = 0, ir al siguiente paso y volver al paso 1
    - Sí. Seleccionar para sustituir e incrementar posición



Influye más la cantidad de memoria principal disponible que el algoritmo de sustitución usado.

EL comportamiento de los programas viene definido por la secuencia de referencias a página que realiza el proceso. Es importante para maximizar el rendimiento del sistema de memoria virtual (TLB, algoritmos de sustitución, ...).

Propiedad de localidad

Hay distintos tipos:

- **Temporal**: una posición de memoria referenciada recientemente tiene una probabilidad alta de ser referenciada en un futuro próximo (ciclos, rutinas, variables globales, ...)
- **Espacial**: si cierta posición de memoria ha sido referenciada es altamente probable que las adyacentes también lo sean (array, ejecución secuencial, ...)

Conjunto de trabajo

Mientras el conjunto de páginas necesarias puedan residir en memoria principal, el número de faltas de página no crece mucho. Si eliminamos de memoria principal páginas de ese conjunto, la activación de paginación crece mucho.

El **conjunto de trabajo** (Working Set) de un proceso es el conjunto de páginas que son referenciadas frecuentemente en un determinado intervalo de tiempo.

$$WS(t,z) = págs\ referenciadas\ en\ el\ intervalo\ de\ tiempo\ t-z\ y\ t$$

Propiedades de los conjuntos de trabajo:

- Los conjuntos de trabajo son **transitorios**, se alternan periodos con un número de páginas relativamente estable (por la propiedad de localidad) con periodos de cambios rápidos en dicho número (cuando el programa se desplaza a una nueva ubicación).
- El pasado no siempre predice el futuro. Tanto el tamaño como el contenido del conjunto de trabajo cambiarán con el tiempo y no podemos predecirlo.
- Difieren unos de otros sustancialmente.

Teoría del Conjunto de Trabajo

Un proceso solo puede ejecutarse si su conjunto de trabajo está en memoria principal.

Una página no puede retirarse de memoria principal si está dentro del conjunto de trabajo del proceso en ejecución.

Hiperpaginación

Si un proceso no tiene suficientes páginas se produce:

- $\shortdownarrow$ uso de la CPU
- $\shortuparrow$ grado de multiprogramación
- $\shortuparrow$ faltas de página

La **hiperpaginación** es la situación en que el SO pasa más tiempo resolviendo faltas de página que ejecutando el programa.

El **grado de multiprogramación** es un factor importante cuando hablamos de hiperpaginación. Si, en un instante dado, hay pocos procesos residentes en memoria, habrá muchas ocasiones en las que todos los

procesos estén bloqueados y se gastará mucho tiempo en el intercambio. Por otro lado, si hay demasiados procesos residentes, el tamaño medio del conjunto residente de cada proceso no será adecuado y se producirán frecuentes faltas de página. El resultado es la hiperpaginación.

Formas de evitar la hiperpaginación

- **Algoritmos de regulación de carga**: actúan directamente sobre el grado de multiprogramación
- **Algoritmos de asignación variables**: aseguran que cada proceso existente tiene asignado un espacio en relación a su comportamiento (los algoritmos expuestos a continuación son de este tipo)

Algortimo basado en el modelo del WS

En cada referencia, determina el conjunto de trabajo: páginas referenciadas en el intervalor (t-x,t] y solo esas páginas son mantenidas en memoria principal.

Algoritmo FFP (Frecuencia de Falta de Página)

Para ajustar el conjunto de páginas de un proceso, usa los intervalos de tiempo entre dos faltas de página consecutivas.

Si el intervalo de tiempo es grande, mayor que Y, todas las páginas no referenciadas en dicho intervalo son retiradas de memoria principal. En otro caso, la nueva página es simplemente incluida en el conjunto de páginas residentes.

Se garantiza que el conjunto de páginas crece cuando las faltas de página son frecuentes y decrece cuando no lo son.

3.4. Gestión de memoria en Linux

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;  //Contador de referencias a esa página
    struct address_space *mapping; //Espacio de direcciones c
    void *virtual; //Dirección virtual si no está libre
    ...
}
```

Una página puede ser utilizada por:

- La caché de páginas. El campo mapping apunta al objeto representado por `address_space`
- Datos privados
- 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

El SO crea unas interfaces para trabajar con la memoria

Interfaces de asignación

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

La función asigna 2^{order} páginas físicas contiguas. Devuelve un puntero a la struct page de la primera página. En caso de error devuelve NULL.

```
unsigned long __get_free_pages(gfp_t_mask, unsigned int order)
```

Esta función asignación 2^{order} páginas físicas contiguas. Devuelve la dirección lógica de la primera página.

```
void * kmalloc(size_t size, gfp_t flags)
```

Asigna size bytes de memoria continua (llama a get_free_pages)

Interfaces de liberación

```
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.

```
void kfree(const void *ptr)
```

Libera memoria de forma similar a `free()` en el espacio de usuario.

Zonas de memoria

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`)
- Modificadores de zona (`GFP_DMA`)
- Tipos (`GFP_KERNEL, GFP_USER`)

Ejemplos 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 (2³) conti
pages */
free_pages(page, 3);
/* our pages are now free and we should no longer access the
address stored in ‘page’
*/
```

Asignación/liberación de memoria en bytes

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

Caché de bloques

La asignación y liberación de estructuras de datos es una de las operaciones más comunes en un kernel de SO. Para agilizar esta solucitud/liberación de memoria Linux usa el **nivel de bloques** (slab layer).

El nivel de bloques actua como un nivel de caché de estructuras genérico que nos permite minimizar el tiempo de creación y liberación. Existe una caché para cada tipo de estructura distinta (ej: task_struct caché o inode caché). Cada caché contiene múltiples bloques constituidos por una o más páginas físicas contiguas. Cada bloque aloja estructuras del tipo correspondiente a la caché. Puede estar en tres estados: lleno, parcial o vacío.

Cuando el kernel solicita una nueva estructura la solicitud se satisface desde un bloque parcial 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 en este nuevo bloque.

El espacio de direcciones de proceso

El **espacio de direcciones de un proceso** (que se encuentre en modo usuario) constituye un espacio de memoria de 32 o 64 bits único. Aunque se puede compartir (`CLONE_VM` para hebras). Linux utiliza memoria virtual (VM). El proceso solo tiene permiso para acceder a determinados intervalos de direcciones de memoria, denominados **áreas de memoria**.

Un área de memoria puede contener:

- 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

El espacio de direcciones de proceso se representa en Linux mediante el **descriptor de memoria**

```
struct mm_struct {
    struct vm_area_struct *mmap; //Lista de áreas de memoria
    struct rb_root mm_rb //Árbol red-black de VMAs, para busc
    struct list_head mmlist; //Lista con todas las mm_struct.
    atomic_t mm_users //Número de procesos utilizando este es
    atomic_t mm_count /*Contador que se activa con la primera
    de direcciones y se desactiva cuando mm_users vale 0*/

    //Límites de las secciones principales
    unsigned long start_code; //Primera dirección de código
    unsigned long end_code; //Última dirección de código
    unsigned long start_data; //Primera dirección de datos
    unsigned long end_data; //Última dirección de datos
    unsigned long start_brk; //Primera dirección de heap
    unsigned long brk; //Última dirección de heap
    unsigned long start_stack; //Primera dirección de la pila
    unsigned long arg_start; //Principio de los argumentos
    unsigned long arg_end; //Final de los argumentos
    unsigned long env_start; //Principio del ámbito del proce
    unsigned long env_end; //Final del ámbito de proceso

    //Información relacionada con las páginas
    pgd_t *pgd //Directorio global de páginas
    unsigned long rss; //Páginas contenidas
    unsigned long total_vm; //Número de páginas totales
}
```

Para **asignar un descriptor de memoria** hay que copiar del descriptor de memoria al ejecutar `fork()`. Compartir el descriptor de memoria mediante el flag `CLONE_VM` de la llamada `clone()`.

Para **liberar un descriptor de memoria** el núcleo decrementa el contador `mm_users` incluido en el `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.

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
    de direcciones*/
    unsigned long vm_start; //Principio del VMA (inclusivo)
    unsigned long vm_end; //Final del VMA (exclusivo)
    unsigned long vm_flagsM //Flags
    struct vm_operations_struct *vm_ops; //Operaciones asocia
    struct vm_area_struct *vm_next; //Lista de VMAs
    struct rb_node vm_rb; //Nodos VMAs en el árbol
};
```

Utilizando `/proc/<pid>/maps` podemos ver las VMAs de un determinado proceso. El formato de 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` : nombre del archivo.\

Para **crear un intervalo de direcciones** utilizaremos `do_mmap()` que nos permite *expandir* un VMA ya existente (porque el intervalo que se añade es adyacente a uno ya existente y tiene los mismos permisos) o *crear* una nueva VMA que represente el nuevo intervalo de direcciones.

```
unsigned long do_mmap(struct file *file, unsigned long addr,
                      unsigned long prot, unsigned long flag, ur
```

Para **eliminar un intervalo de direcciones** empleamos `do_munmap()`. El parámetro `mm` especifica el 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
```

Las **direcciones virtuales** deben convertirse a direcciones físicas mediante tablas de páginas. Linux hace uso de una estructura de tabla de páginas con tres niveles, formada por los siguientes tipos de tablas:

- La tabla 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`. Cada entrada en el directorio de páginas señala a una página del directorio intermedio de páginas. Para un proceso activo, el PGD tiene que estar en la memoria principal.
- La tabla intermedia será el **directorio intermedio de páginas** (Page Middle Directory, PMD), que es un array de tipo `pmd_t`. Cada array de este directorio señala a una página de la tabla de páginas.
- El último nivel es la **tabla de páginas** (Page Table Entry, PTE) y contiene entradas de la tabla de páginas del tipo `pte_t` que apuntan a páginas físicas: `struct_page`

Para utilizar esta estructura de la tabla de páginas a tres niveles, una dirección virtual en Linux se ve como un conjunto de cuatro campos. El campo más a la izquierda se utiliza como índice en el directorio de páginas. El siguiente campo sirve como índice en el directorio intermedio de páginas. El tercer campo sirve como índice en la tabla de páginas. El cuarto campo indica el desplazamiento dentro de la página seleccionada en memoria.

La caché de páginas y la escritura de páginas a disco

La caché de páginas está constituida por páginas físicas de RAM cuyos contenidos 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). Esta caché realiza la lectura/escritura de datos de/a disco. Hay varias fuentes de datos para la caché: archivos regulares, de dispositivos y archivos proyectados en memoria.

Desalojo de la caché de páginas

Es el proceso por el cual se eliminan datos de la caché junto con la estrategia para decidir qué 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 tener disponibles más páginas limpias. Ahora queda por decidir qué 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 más antiguo. El problema es el acceso a archivos una sola vez.

Linux soluciona el problema usando dos listas pseudo-LRU: *active list* e *inactive list*. Las páginas de la active list no pueden ser seleccionadas como víctimas, las páginas de la inactive list sí.

Operaciones

Una página puede contener varios bloques de disco posiblemente no contiguos (depende del método de asignación de bloques a archivos). La caché de páginas de Linux usa una estructura para gestionar entradas de la caché y operaciones de E/S de páginas: address_space. Se realiza la lectura/escritura de páginas de/en caché. Hebras de escritura retardada.