Eloy García Martínez

Alfredo López López

- El concepto de paginación aparece junto al de memoria virtual.
- La memoria virtual, es una técnica que permite al software utilizar más memoria que la que realmente existe.
- La paginación, se encarga del intercambio de información entre nuestra memoria y el disco duro.l
- Linux usa la paginación bajo demanda para tener en memoria sólo las páginas que están siendo usadas.

- Existen dos estructuras relacionadas con la memoria:

  - ***vm_area_struct***, representa un área de memoria virtual.

  - ***mm_struct,*** describe el contenido de la memoria virtual.

Diagrama general

task_struct

mm

mm_struct

count

pgd

mmap

mmap_avl

mmap_se
m

vm_area_struct

vm_end

vm_start

vm_flags

vm_inode

vm_ops

vm_next

vm_area_struct

vm_end

vm_start

vm_flags

vm_inode

vm_ops

vm_next

DATA

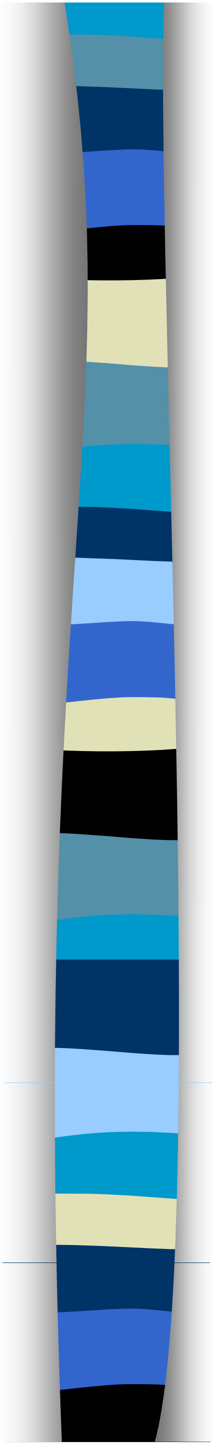
CODE

Memoria virtual

0x8059BB8

0x8048000

0x0000000



VM_AREA

```
struct vm_area_struct {
       struct mm_struct * vm_mm; /*la dirección de memoria a la que pertenece
       unsigned long vm_start;  /*dirección de inicio
       unsigned long vm_end;  /*dirección de fin
       struct vm_area_struct *vm_next;  /*puntero que apunta a la siguiente vm_area_struct del proceso
       pgprot_t vm_page_prot;  /*protección de la página
       unsigned long vm_flags;  /*estado
       struct rb_node vm_rb;
       union {
              struct {
                     struct list_head list;
                     void *parent;
                     struct vm_area_struct *head;
              } vm_set;

              struct raw_prio_tree_node prio_tree_node;
      } shared;
                ...
```

```
struct list_head anon_vma_node;
struct anon_vma *anon_vma;
struct vm_operations_struct * vm_ops;  /*operaciones que tiene asociadas

unsigned long vm_pgoff;
struct file * vm_file;
void * vm_private_data;
unsigned long vm_truncate_count;

#ifndef CONFIG_MMU
        atomic_t vm_usage;
#endif
#ifdef CONFIG_NUMA
        struct mempolicy *vm_policy;
#endif
};
```

mm-struct

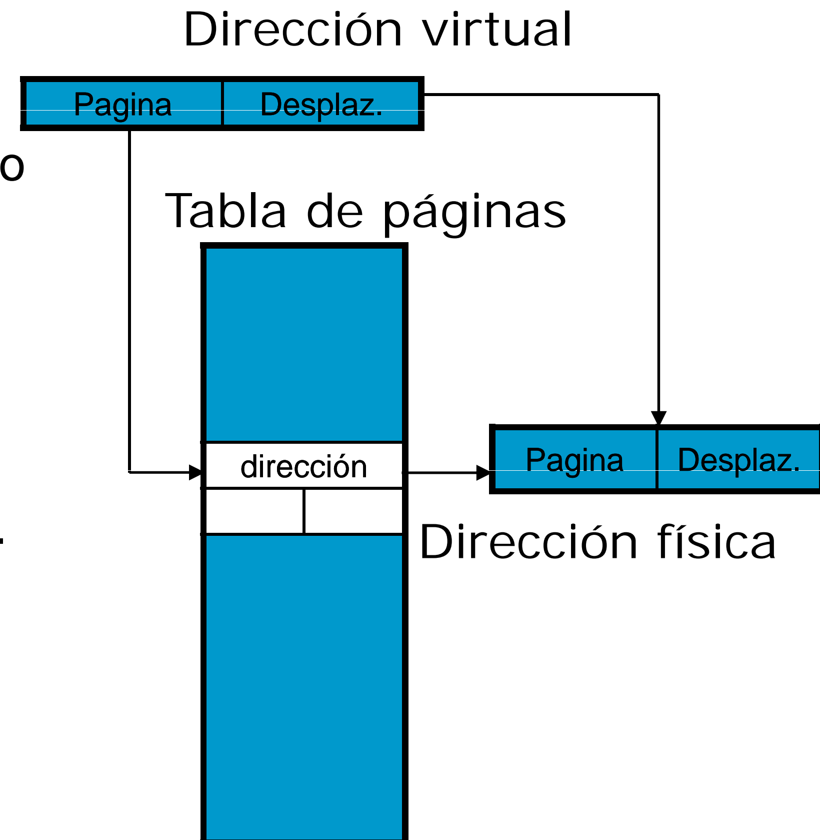
```
struct mm_struct {
    struct vm_area_struct * mmap; /* Lista de VMAs
    struct rb_root mm_rb;
    struct vm_area_struct * mmap_cache;
    unsigned long (*get_unmapped_area) (struct file *filp,
                    unsigned long addr, unsigned long len,
                    unsigned long pgoff, unsigned long flags);
    void (*unmap_area) (struct mm_struct *mm, unsigned long addr);
    unsigned long mmap_base;  /* base del área mmap
    unsigned long task_size;   /* tamaño de la memoria virtual del proceso
    unsigned long cached_hole_size;
    unsigned long free_area_cache;
    pgd_t * pgd;
    atomic_t mm_users;   /* número de usuarios
    atomic_t mm_count;   /* referencias al “struct”
    int map_count;
    struct rw_semaphore mmap_sem;  /* protección de la tabla de página
    spinlock_t page_table_lock;
    struct list_head mmlist;
    mm_counter_t _file_rss;
    mm_counter_t _anon_rss;
    unsigned long hiwater_rss;
    unsigned long hiwater_vm;
```

```
        ...
    unsigned long total_vm, locked_vm, shared_vm, exec_vm;
    unsigned long stack_vm, reserved_vm, def_flags, nr_ptes;
    unsigned long start_code, end_code, start_data, end_data;
    unsigned long start_brk, brk, start_stack;
    unsigned long arg_start, arg_end, env_start, env_end;
    unsigned long saved_auxv[AT_VECTOR_SIZE];
    cpumask_t cpu_vm_mask;
    mm_context_t context;
    unsigned int faultstamp;
    unsigned int token_priority;
    unsigned int last_interval;
    unsigned long flags;
    struct core_state *core_state;
    rwlock_t            ioctx_list_lock;
    struct kioctx        *ioctx_list;
#ifdef CONFIG_MM_OWNER
    struct task_struct *owner;
#endif
#ifdef CONFIG_PROC_FS
     struct file *exe_file;
     unsigned long num_exe_file_vmas;
#endif
#ifdef CONFIG_MMU_NOTIFIER
    struct mmu_notifier_mm *mmu_notifier_mm;
#endif};
#endif
```

Tabla de paginas I

- Todas las direcciones que lanza el procesador son direcciones virtuales y no direcciones físicas.
- Dirección virtual se compone de:
  - Nº de página virtual
  - Desplazamiento dentro de la página.
- Páginas compartidas por 2 procesos.
- Número de marco de página único.
- Información de la tabla de páginas:
  - Flag de validación
  - Nº de marco
  - Información de control.

Estructura page (<linux/mm_types.h>)



Estructura de una página

(<linux/mm_types.h>)

```
struct page {
        unsigned long flags;                    ← Estado de la página
        atomic_t _count;                        ← Números de referencia a la página
        union {
                atomic_t _mapcount;             ← Número de entradas de tablas de página referidas al marco de página
                struct {
                        u16 inuse;              ← Número de objetos
                        u16 objects;
        };};
        union {
            struct {
                unsigned long private;          ← Cuando la página está libre indica el orden en el buddy system
                struct address_space *mapping;  ← Si es NULL apunta a inode, si no apunta a anon_vma
            };
```

Estructura de una página II

```
#if USE_SPLIT_PTLOCKS
            spinlock_t ptl;
#endif
            struct kmem_cache *slab;
            struct page *first_page;
        };
        union {
            pgoff_t index;
            void *freelist;
        };
        struct list_head lru;
#if defined(WANT_PAGE_VIRTUAL)
        void *virtual;
#endif
};
```

Desplazamiento dentro del mapeado

Lista de páginas a desalojar mediante algoritmo LRU

Posibles estados de la página

(<linux/page-flags.h>)

#define PG_locked 0 — La página está bloqueada en memoria

#define PG_error 1 — Se ha producido un error en la carga de la página

#define PG_referenced 2 — La página ha sido Accedida

#define PG_uptodate 3 — El contenido de la página está actualizado

#define PG_dirty 4 — Indica si el contenido de la página se ha modificado

#define PG_lru 5 — La página está en la lista de páginas activas o inactivas

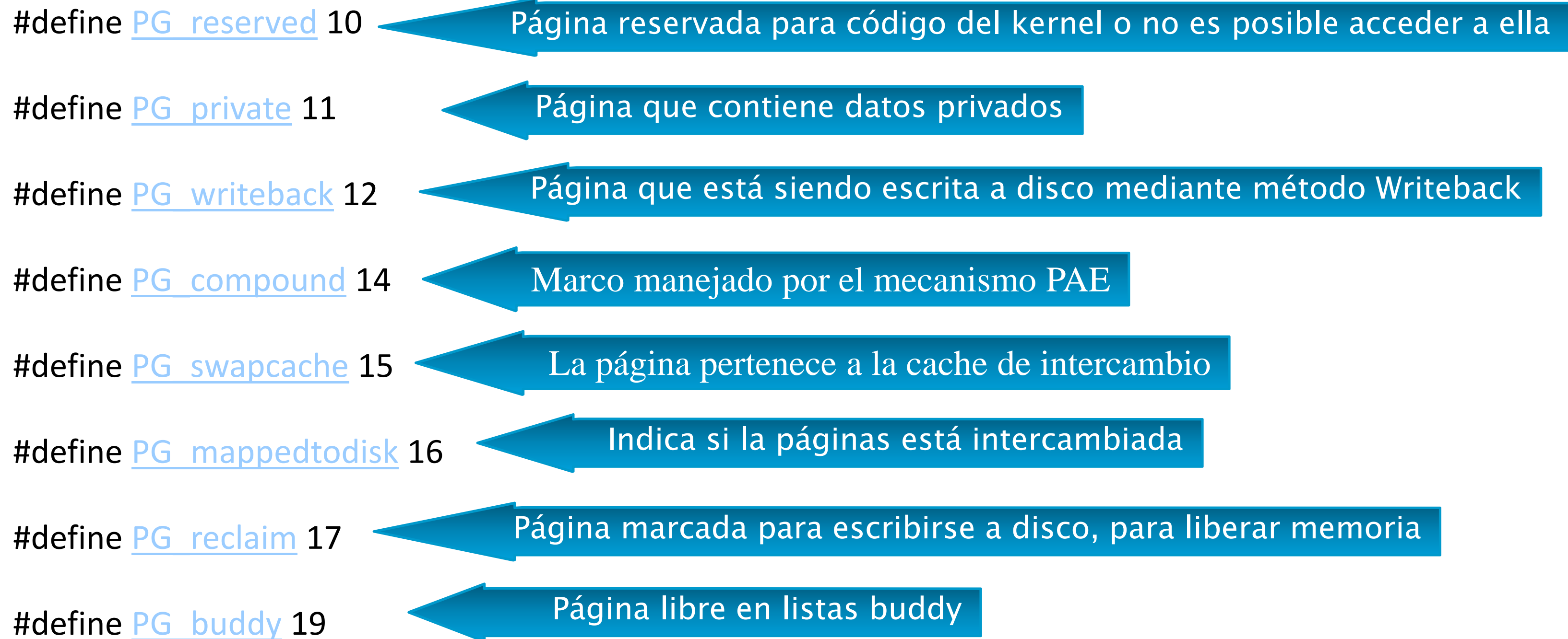
#define PG_active 6 — La página está en la lista de páginas activas

#define PG_slab 7 — El marco de página está incluido en un slab

#define PG_owner_priv_1

#define PG_arch_1 9 — No se utiliza la arquitectura 80x86

Posibles estados de la página II



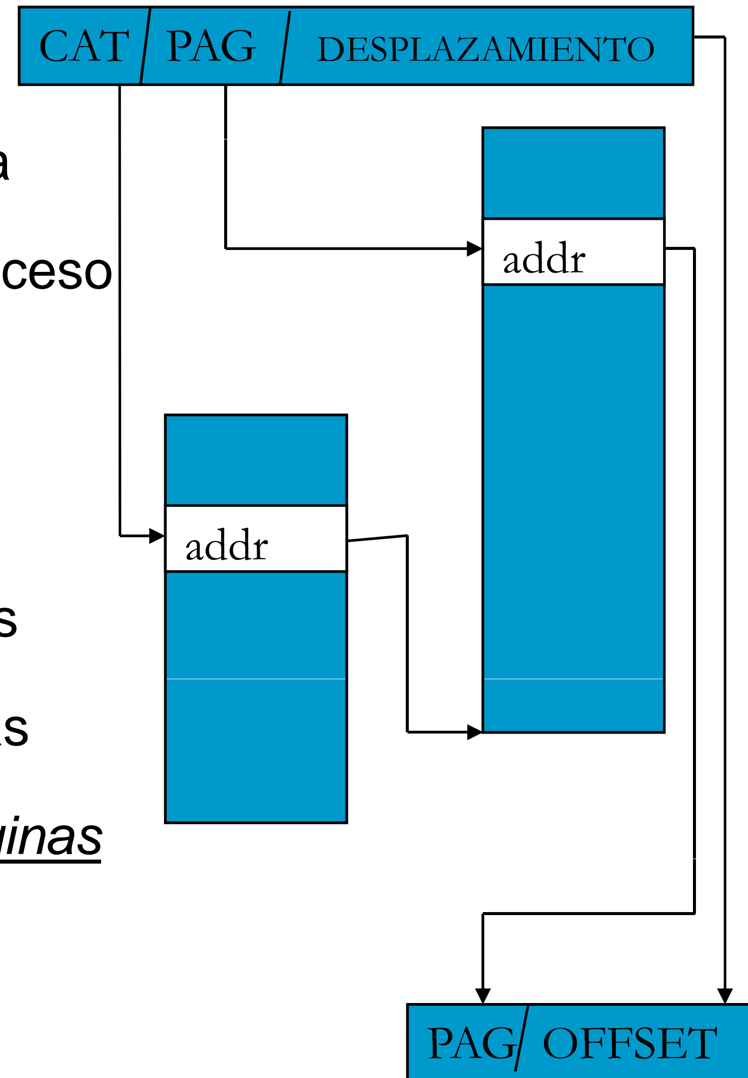
Problemas de la tabla de página

– Mantener la tabla en memoria

– Ralentiza la ejecución del proceso

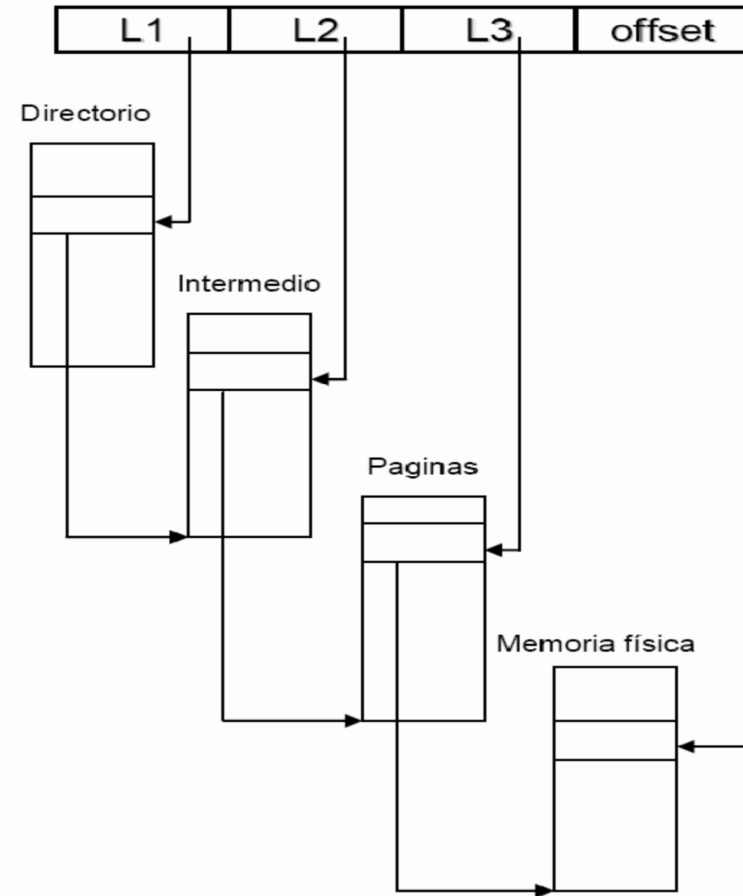
- Solución:

  - Linux descompone la tabla de páginas original en distintos niveles, este conjunto de tablas se denomina *directorio de páginas*



El DTP

- Dirección Base (10 bits)
- Índice (10 bits)
- Desplazamiento dentro del marco de página
- Arquitectura basada en 3 niveles.
- Tipos de datos



Las siguientes estructuras de datos muestran los 3 níveles existentes:

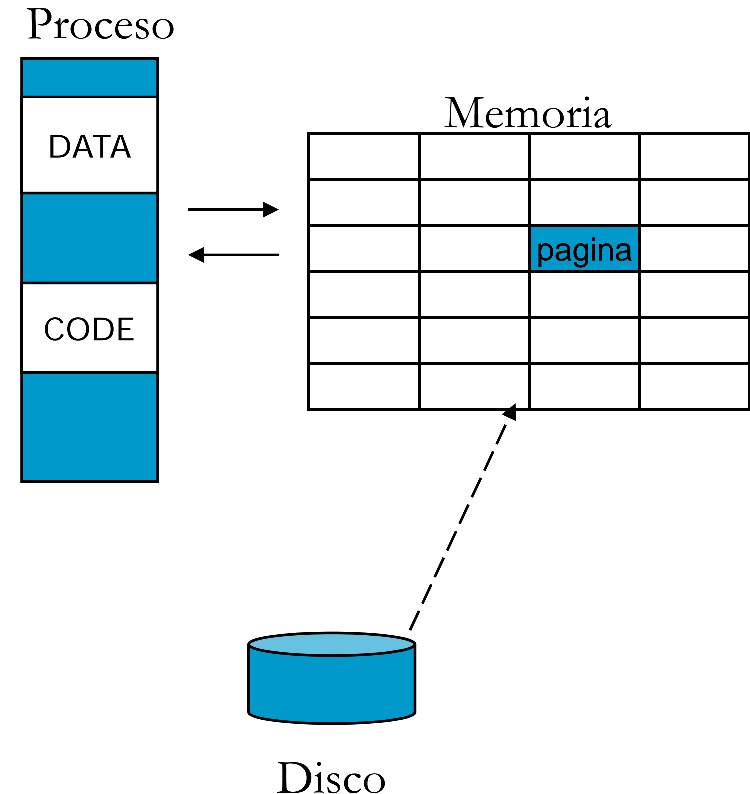
typedef struct { unsigned long pte_low, pte_high; } pte_t;

typedef struct { unsigned long long pmd; } pmd_t;

typedef struct { unsigned long long pgd; } pgd_t;

Caché de páginas

- Mejora del rendimiento
- Linux emplea diferentes caches para la gestión de la memoria:
  - Buffer Cache
  - Cache de Páginas
  - Cache de Intercambio (swap)
  - Caches Hardware
- La caché se gestiona dinámicamente.



Asignación y liberación de páginas

- Buddy System
  - ¿Qué es?
  - **“El núcleo mantiene listas de grupos de páginas *disponibles, una por nivel*.”**
  - Peticiones en bloques de páginas potencia de 2.
  - ¿Cómo funciona?
    - Peticiones.
    - Liberaciones.

- Se buscan bloques de páginas contiguas del mismo tamaño pedido. A partir de una estructura free_list.
- Si se encuentra un bloque libre se devuelve, si no, buscamos el de orden inmediatamente superior.
- Una vez encontrado el de orden superior, se trocea hasta llegar al nivel de bloque pedido.

- El resto de los bloques troceados se introducen en free_list.
- En free_list se comprueban si los grupos de páginas adyacentes están libres.
- Si es así, se fusionan y se introducen en la lista de nivel adecuado.

Asignación y liberación de páginas

La tabla *free_area* (linux/mmzone.h) contiene la dirección del primer bloque de páginas libre por nivel:

```
struct free_area {
        struct list_head free_list[MIGRATE_TYPES];
        unsigned long nr_free;
};
```

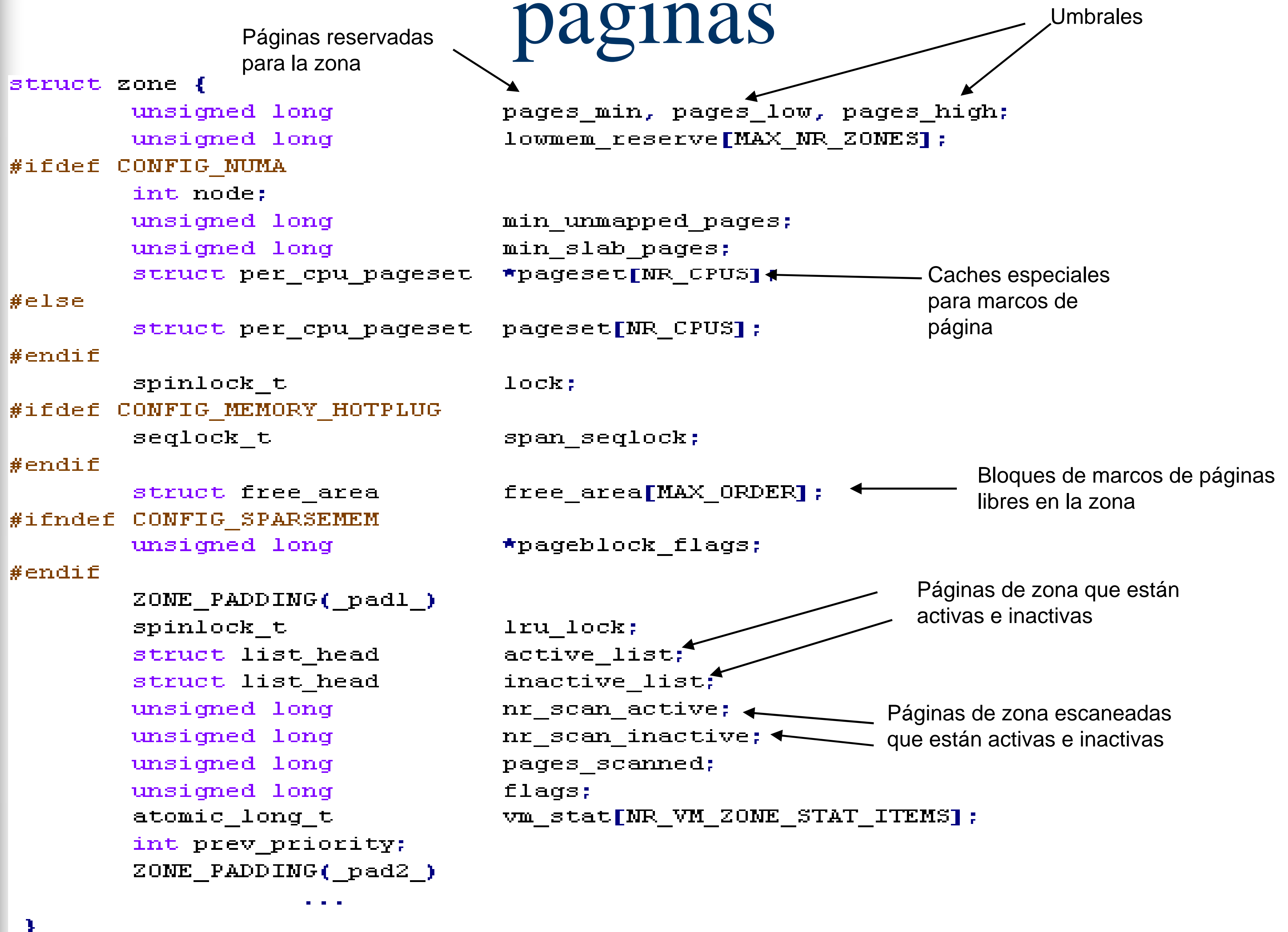
La estructura zonelist (linux/mmzone.h) contiene un vector de estructuras zones:

```
struct zonelist {
        struct zonelist_cache *zlcache_ptr;
        struct zone *zones[MAX_ZONES_PER_ZONELIST + 1];
#ifdef CONFIG_NUMA
        struct zonelist_cache zlcache;
#endif
};
```

La memoria física se divide en multiples zonas:

0- ZONE_DMA < 16 MB ISA DMA capable memory
1- ZONE_NORMAL 16-900 MB direct mapped by the kernel
2- ZONE_HIGHMEM > 900 MB only page cache and user processes
3- MAX_NR_ZONES

Asignación y liberación de páginas



Asignación y liberación de páginas

Obtiene un único marco de página y lo rellena a 0,s

Get_zeroed_page devuelve una dirección de 32 bits

```
unsigned long get_zeroed_page(gfp_t gfp_mask)
{
        struct page * page;
        VM_BUG_ON((gfp_mask & __GFP_HIGHMEM) != 0);
        page = alloc_pages(gfp_mask | __GFP_ZERO, 0);
        if (page)
                return (unsigned long) page_address(page);
        return 0;
}
```

```
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)
{
        struct page * page;
        page = alloc_pages(gfp_mask, order);
        if (!page)
                return 0;
        return (unsigned long) page_address(page);
}
```

Intenta conseguir páginas libres

Asignación y liberación de páginas

(<linux/mm/page_alloc.c>)

```
void __free_pages(struct page *page, unsigned int order)
{
        if (put_page_testzero(page)) {
                if (order == 0)
                        free_hot_page(page);
                else
                        __free_pages_ok(page, order);
        }
}

EXPORT_SYMBOL(__free_pages);

void free_pages(unsigned long addr, unsigned int order)
{
        if (addr != 0) {
                VM_BUG_ON(!virt_addr_valid((void *)addr));
                __free_pages(virt_to_page((void *)addr), order);
        }
}
```

Página en caché

Página en memoria principal

Libera página

- **El NÚCLEO**

- **Linux ofrece 2 tipos de funciones para asignar zonas de memoria del espacio de direccionamiento propio del núcleo:**
  - ***kmalloc* y *kfree*: páginas contiguas en memoria central**
  - ***vmalloc* y *vfree*: páginas no contiguas en memoria central**

Kmalloc

(<linux/mm/slab_def.h>)

```c
static inline void *kmalloc(size_t size, gfp_t flags)
{
        if (__builtin_constant_p(size)) {
                int i = 0;

                if (!size)
                        return ZERO_SIZE_PTR;

#define CACHE(x) \
                if (size <= x) \
                        goto found; \
                else \
                        i++;
#include <linux/kmalloc_sizes.h>
#undef CACHE
                {
                        extern void __you_cannot_kmalloc_that_much(void);
                        __you_cannot_kmalloc_that_much();
                }
found:
#ifdef CONFIG_ZONE_DMA
                if (flags & GFP_DMA)
                        return kmem_cache_alloc(malloc_sizes[i].cs_dmacachep,
                                                flags);
#endif
                return kmem_cache_alloc(malloc_sizes[i].cs_cachep, flags);
        }
        return __kmalloc(size, flags);
}
```

Si la variable en el tiempo de compilación es conocida como una constante

Coge memoria

Kfree

(<linux/mm/slab.c>)

```
void kfree(const void *objp)
{
        struct kmem_cache *c;
        unsigned long flags;

        if (unlikely(ZERO_OR_NULL_PTR(objp)))
                return;
        local_irq_save(flags);
        kfree_debugcheck(objp);
        c = virt_to_cache(objp);
        debug_check_no_locks_freed(objp, obj_size(c));
        debug_check_no_obj_freed(objp, obj_size(c));
        __cache_free(c, (void *)objp);
        local_irq_restore(flags);
}
```

Salva el estado de la CPU actual y deshabilita interrupciones

vmalloc_node

(<linux/mm/vmalloc.c>)

```
static void *__vmalloc_node(unsigned long size, gfp_t gfp_mask, pgprot_t prot,
                            int node, void *caller)
{
        struct vm_struct *area;

        size = PAGE_ALIGN(size);
        if (!size || (size >> PAGE_SHIFT) > num_physpages)
                return NULL;

        area = __get_vm_area_node(size, VM_ALLOC, VMALLOC_START, VMALLOC_END,
                                  node, gfp_mask, caller);

        if (!area)
                return NULL;

        return __vmalloc_area_node(area, gfp_mask, prot, node, caller);
}

void *__vmalloc(unsigned long size, gfp_t gfp_mask, pgprot_t prot)
{
        return __vmalloc_node(size, gfp_mask, prot, -1,
                              __builtin_return_address(0));
}

void *vmalloc(unsigned long size)
{
        return __vmalloc_node(size, GFP_KERNEL | __GFP_HIGHMEM, PAGE_KERNEL,
                              -1, __builtin_return_address(0));
}
```

Comprueba que el tamaño es 0 o mayor que el permitido

Intenta alojar un área de memoria del tamaño especificado

Retornamos la dirección de la nueva zona de memoria

__vunmap (<linux/mm/vmalloc.c>)

```
static void __vunmap(const void *addr, int deallocate_pages)
{
        struct vm_struct *area;

        if (!addr)
                return;

        if ((PAGE_SIZE-1) & (unsigned long)addr) {
                WARN(1, KERN_ERR "Trying to vfree() bad address (%p)\n", addr);
                return;
        }

        area = remove_vm_area(addr);
        if (unlikely(!area)) {
                WARN(1, KERN_ERR "Trying to vfree() nonexistent vm area (%p)\n",
                                addr);
                return;
        }

        debug_check_no_locks_freed(addr, area->size);
        debug_check_no_obj_freed(addr, area->size);

        if (deallocate_pages) {
                int i;

                for (i = 0; i < area->nr_pages; i++) {
                        struct page *page = area->pages[i];

                        BUG_ON(!page);
                        __free_page(page);
                }

                if (area->flags & VM_VPAGES)
                        vfree(area->pages);
                else
                        kfree(area->pages);
        }

        kfree(area);
        return;
}
```

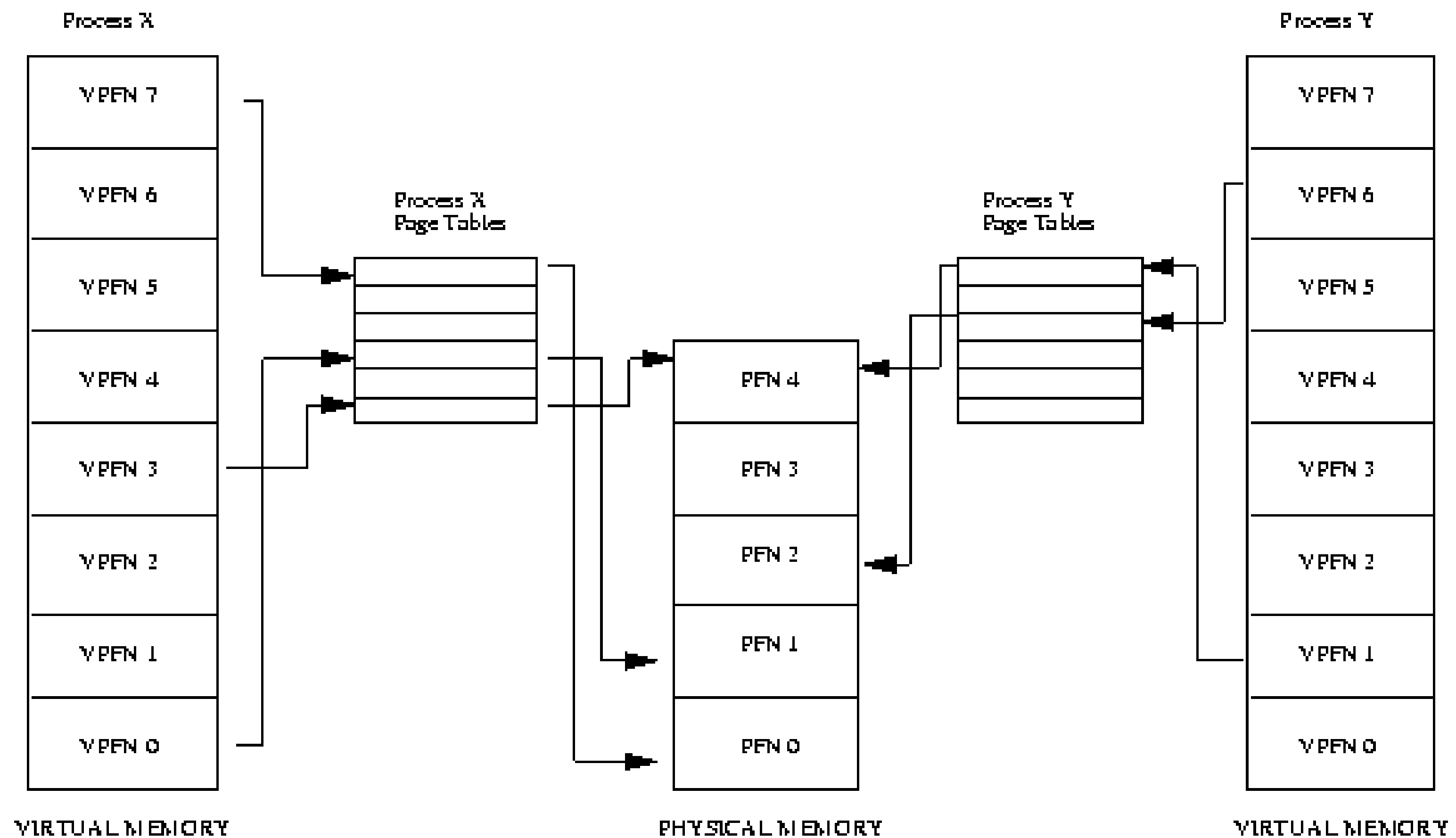
Dirección incorrecta, no es múltiplo del tamaño de página

Borra el área de memoria apuntada por la addr

```
void vfree(const void *addr)
{
        BUG_ON(in_interrupt());
        __vunmap(addr, 1);
}
```

Copy and Write

- Copy-on-write es una técnica para realizar eficientemente la copia de páginas.



- Tipos de fallos:
  - Acceso incompatible con la protección asociada a una página en memoria.

Las funciones usadas en el tratamiento de una excepción se definen en mm/memory.c:

- do_wp_page
- do_swap_page
- handle_pte_fault
- handle_mm_fault
- do_page_fault

Fallos de página II

La función do_wp_page gestiona la copia en escritura.

Cuando un proceso accede en escritura a una página compartida y protegida en lectura exclusiva, se asigna una nueva página, y se comprueba si la página afectada es compartida por varios procesos.

En caso afirmativo se copia su contenido en la nueva página, y se inserta en la tabla de páginas del proceso. El número de referencias a la anterior página se decrementa por la llamada a liberar la pagina.

En el caso de que la página afectada no sea compartida, su protección simplemente se modifica para hacer posible la escritura.

- **Handle_mm_fault**:
  - Esta función es invocada por do_page_fault, cuando una página a la que se intenta acceder no está en memoria principal.
  - Valores devueltos:

1. VM_MINOR_FAULT: El proceso no se duerme, la página está en caché.
2. VM_MAJOR_FAULT: El proceso debe dormirse en lo que se transfiere la página (presumiblemente de disco)
3. VM_FAULT_OOM: No hay memoria disponible

- **Handle_pte_fault**:
  – Función que inspecciona la entrada de tabla de páginas para determinar qué ocasionó el fallo y llamar al manejador adecuado.

- **Do_swap_page**:
  – Carga en memoria el contenido de una página situada en el espacio swap.
  – Puede devolver los siguientes valores:

  1: La página ya estaba en la swap cache.

  2: La página tuvo que leerse del área de swap.

  -1: Hubo algún problema en la transferencia de la página.

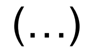
Fallos de página VI

do_page_fault se encarga de manejar los fallos de página, se encuentra en el archivo /arch/x86/mm/fault_32.c
La funcionalidad en resumen es:
1- Determinar la dirección de memoria que ocasionó la interrupción.
2- Determinar el problema y llamar a la rutina apropiada.
Si se trata de un acceso indebido el núcleo avisa al proceso mediante una señal.
Puede ocurrir que sea el núcleo el que generó la interrupción.

```
/*
* Esta rutina se encarga de manejar los fallos de pagina, determina la
* dirección y el problema, llamando a la rutina apropiada
* error_code:
*     bit 0 == 0 página no encontrada, 1 significa fallo de protección
*     bit 1 == 0 lectura, 1 escritura
*     bit 2 == 0 modo kernel, 1 modo usuario
*     bit 3 == 1 uso del bit reservado detectado
*     bit 4 == 1 el fallo se produjo en la búsqueda de una instrucción
*/
fastcall void __kprobes do_page_fault(struct pt_regs *regs,
                        unsigned long error_code)
```

Do_fault_page

```
void __kprobes do_page_fault(struct pt_regs *regs, unsigned long error_code){
        /*variables
        /*tomamos el proceso actual su información de gestión de memoria
          tsk = current;
          mm = tsk->mm;
          prefetchw(&mm->mmap_sem);
        /*cogemos la dirección que provocó la excepción
          address = read_cr2();
          si_code = SEGV_MAPERR;
        /* hace unas comprobaciones
        /* se obtiene el descriptor de la región de la memoria afectada
        /* y luego comprueba el tipo de error
          vma = find_vma(mm, address);
        /* aqui comprueba el tipo de error
```



```
(…)
/*si llegamos aquí es porque tenemos una buena vm_area, así que podremos
 /* manejar el fallo siempre y cuando no tengamos problemas
 /*con los permisos de lectura y escritura
GOOD_AREA:
        si_code = SEGV_ACCERR;
        write = 0;
        switch (error_code & (PF_PROT|PF_WRITE)) {
                /* se comprueban que se tienen los permisos adecuados
                /* en caso de no tenerlos va a Bad_area
        }
      /* si hemos llegado hasta aquí, implicará el acceso a memoria es correcto
      /* pero la página no estaba en memoria
        fault = handle_mm_fault(mm, vma, address, write);
      /* si se produce algún error iremos a sigbus o out of memory según sea éste
        up_read(&mm->mmap_sem);
        return;
```

/* si se llega aquí hay un fallo de protección.

/* existen dos tipos de error: si intenta escribir en una página

/* que no corresponde al proceso, ó se intenta escribir en una página que si corresponde

/* pero esta protegida contra escritura.

BAD_AREA:

/* se llega a éste punto si es un problema de núcleo

NO_CONTEXT:

/* estamos en un área fuera de la memoria del proceso

OUT_OF_MEMORY:

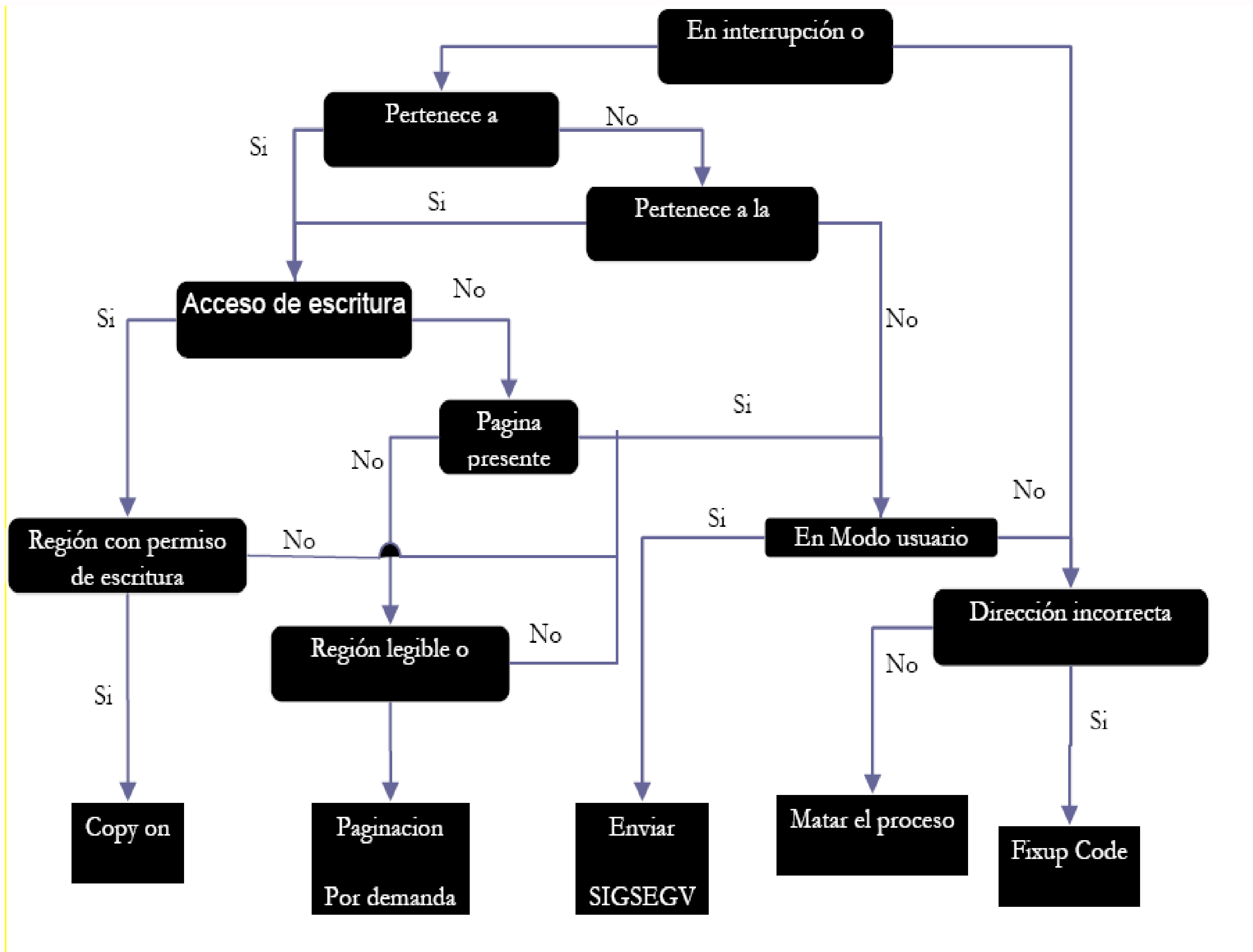
/* si se llega aquí, hubo un fallo de página y no se pudo cargar

/*desde el dispositivo de intercambio.

DO_SIGBUS:

}

RESUMEN:



Eloy García Martínez

Alfredo López López