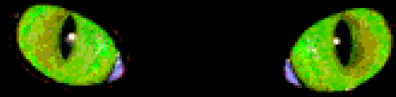
Llamada del sistema BRK

Enrique González Rodríguez
Diego Trujillo García

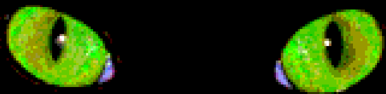


ÍNDICE

- Introducción a la memoria dinámica
- Memoria
- Llamada BRK
- Llamada SBRK
- Uso de BRK y SBRK
- Sys_brk
- Asignación de zonas de memoria
- Llamada VMALLOC
- LLamada GET_VM_AREA
- Llamada VFREE
- Mprotect

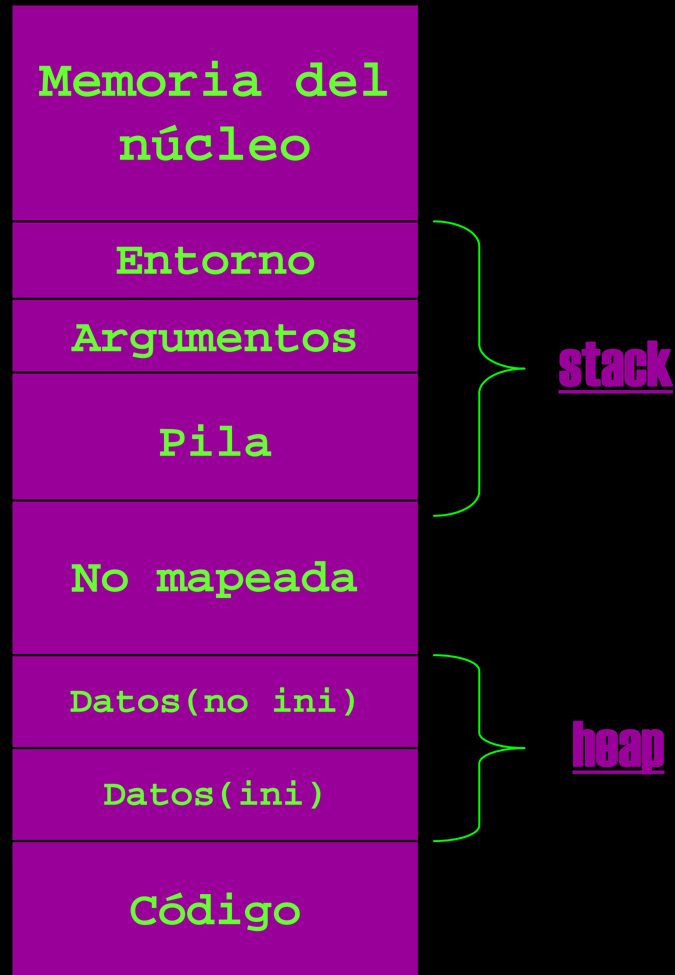
Introducción a la memoria dinámica

- Es la memoria que solicitan los procesos en tiempo de ejecución a medida que la va necesitando
- Para gestionar la memoria se utilizan punteros, ya que no podemos saber a priori donde quedarán libres huecos de memoria

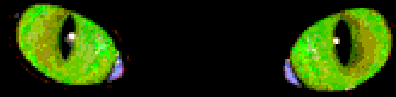


Índice

Memoria



Índice



Llamada BRK

- Se utiliza para modificar el tamaño de la zona heap (segmento de datos).
- La forma de llamarlo es la siguinte

```
int brk(void *end_data_segment)
```

Donde end_data_segment es un puntero que contiene la dirección del final de heap. Esta dirección tiene que ser superior a la dirección del segmento de código e inferior en 16 k al segmento de código de pila.

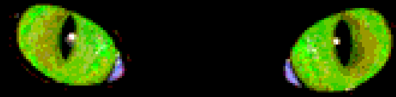
Índice

- Es una llamada derivada de BRK
- Se utiliza para variar el tamaño de los datos
- La forma de llamarlo es:

```
void *sbrk(long size);
```

Si size es mayor que 0, se aumenta el tamaño, si es menor que 0, lo disminuye

Índice

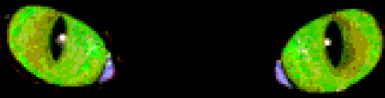


Uso de BRK y SBRK

- BRK y SBRK no están incluidas en el estándar de C y es muy laborioso gestionar la memoria con ellas
- Sin embargo son usadas por otras funciones tales como:

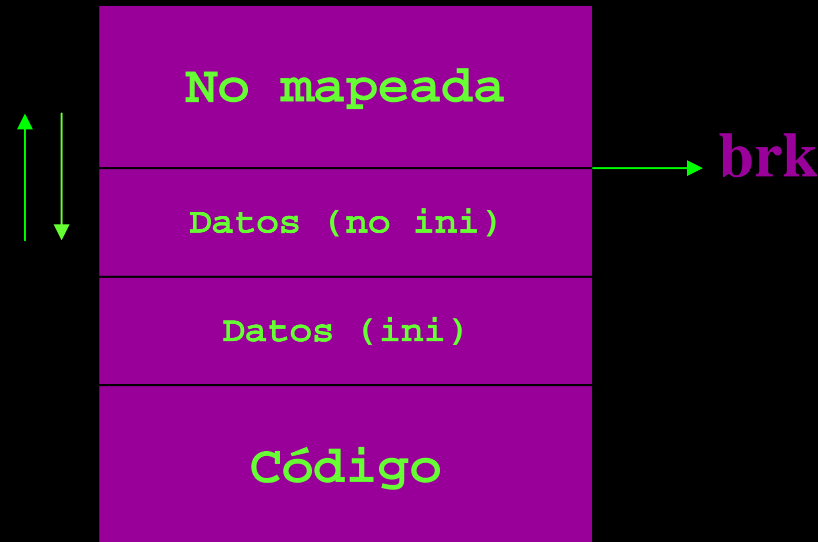
Malloc Calloc Realloc Free

Índice



Sys_brk

- Se encuentra implementada en mm/mmap.c
- Es la llamada que modifica el valor de brk, para poder cambiar el tamaño del segmento de datos



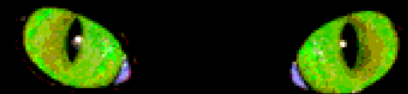
Índice

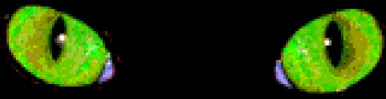
PASOS DEL ALGORITMO

1) Si la nueva dirección de *brk* que se le pasa es inferior a la actual, se desasigna la zona de memoria mediante do_munmap.

2) Si es superior, se asigna memoria con do_brk.

3) Si la nueva dirección es incorrecta (zona de pila o de código), simplemente devuelve el valor actual de brk.

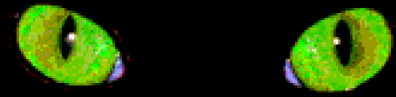




Sys_brk (I)

Cuando un proceso en modo usuario invoca la llamada brk, el kernel ejecuta la función sys_brk(addr).

```
asmlinkage unsigned long sys_brk(unsigned long brk)
{
    unsigned long rlim, retval;
    unsigned long newbrk, oldbrk;
    struct mm_struct *mm = current->mm;
```



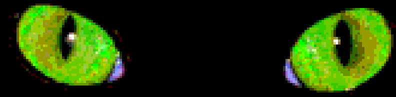
Sys_brk (II)

Se bloquea la escritura en la memoria por parte del proceso actual.

```
down_write(&mm->mmap_sem);
```

Comprobamos que el brk no se coloca en la zona dedicada al código.

```
if (brk < mm->end_code)
     goto out;
```



Sys_brk (III)

la función alinea la dirección en múltiplos indicado por PAGE_SIZE y compara el nuevo brk con el actual.

```
newbrk = PAGE_ALIGN(brk);
oldbrk = PAGE_ALIGN(mm->brk);
if (oldbrk == newbrk)
    goto set_brk;
```

Si el nuevo brk esta por debajo del antiguo, libera el trozo de memoria que hay entre ambos utilizando la función do_munmap()

```
if (brk <= mm->brk) {
    if (!do_munmap(mm, newbrk, oldbrk-
    newbrk))
            goto set_brk;
    goto out; }
```

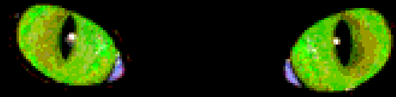
Sys_brk (IV)

Si el proceso quiere aumentar el heap, **sys_brk()** primero chequea si el proceso tiene permiso para hacerlo. Si el proceso esta intentando alojar memoria fuera de sus limites, la función simplemente retorna el valor original de **mm->brk** sin asignar más memoria.

```
rlim = current->signal >rlim[RLIMIT_DATA].rlim_cur;
    if (rlim < RLIM_INFINITY && brk - mm->start_data > rlim)
            goto out;
```

Comprueba que la zona de memoria se puede utilizar, examinando que no se solapa con ninguna otra región de memoria en los márgenes establecidos.

```
if (find_vma_intersection(mm, oldbrk, newbrk+PAGE_SIZE))
    goto out;
```



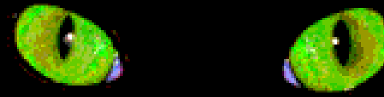
Sys_brk (V)

Si todo ha salido bien, la función do_brk() es invocada. Si esta retorna oldbrk, la asignación fue satisfactoria y sys_brk() devuelve el valor addr, en caso contrario, este retorna el viejo mm->brk.

```
if (do_brk(oldbrk, newbrk-oldbrk) != oldbrk)
      goto out;
set_brk:
mm->brk = brk;
out:
      retval = mm->brk;
```

volvemos a habilitar la escritura y devolvemos retval

```
up_write(&mm->mmap_sem);
return retval;
}
```



Asignación de zonas de memoria

- Linux ofrece varios tipos de funciones que permiten asignar zonas de memoria para la utilización propia del núcleo

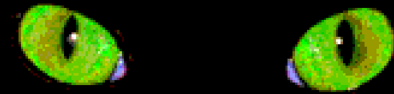
- **kmalloc** y **kfree** permiten asignar y liberar zonas de memoria formadas por páginas contiguas en memoria central. Suelen ser muy útiles para la gestión de zonas de pequeño tamaño, aunque son notoriamente ineficientes para zonas de grandes tamaños, ya que utilizan bloques de tamaño fijo y devuelven un bloque cuyo tamaño puede ser muy superior al tamaño solicitado.

- **vmalloc** y **vfree** permiten asignar y liberar zonas de memoria formadas por páginas que no son forzosamente contiguas en memoria central. No presentan el problema de las otras funciones. Estas páginas se insertan seguidamente en el segmento reservado al núcleo modificando las tablas de páginas. De este modo no se desperdicia tanto el tamaño de la memoria asignado.

Índice

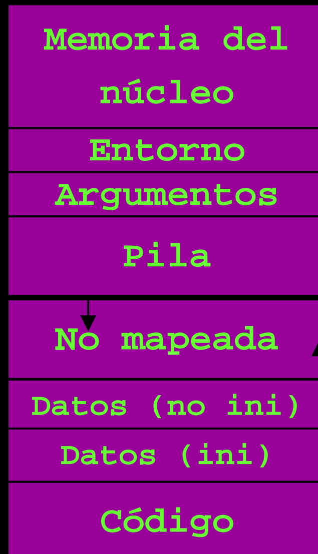
Asignación de zonas de memoria(I)

- La función **vmalloc** trabaja de forma similar a **kmalloc**, excepto que asigna memoria que solamente es contigua virtualmente, no físicamente. Esta es la forma de trabajar de la función de librería malloc de espacio de usuario, se devuelve un trozo de memoria que es contigua dentro del espacio virtual de direcciones del procesador, pero no se garantiza que sea contigua en la RAM física.



VMALLOC_END = 4 GB

VMALLOC_START



- Generalmente los dispositivos hardware necesitan trozos de memoria contigua físicamente, es decir, necesitan usar **kmalloc**, también por rendimiento el kernel suele usar **kmalloc**, se reserva el uso de **vmalloc** a situaciones donde se quiere asignar gran cantidad de memoria (el coste de reservar memoria física contigua en grandes cantidades es mayor)

Llamada VMALLOC

Le pasamos el tamaño de memoria que solicitamos en size

```
void *__vmalloc(unsigned long size, int gfp_mask, pgprot_t prot){
struct vm_struct *area;
```

La dirección donde se encuentran los descriptores de páginas la podemos encontrar en pages.

```
struct page **pages;
unsigned int nr_pages, array_size, i;
```

Alineamos el tamaño que solicitamos y comprobamos si el tamaño es 0 o mayor que el permitido

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

Índice

Mediante el get_vm_area, que comentaremos a continuación, creamos un nuevo descriptor y retornamos la dirección asignada para el área de memoria. Si no se ha podido alojar la memoria, retornamos NULL.

```
area = get_vm_area(size, VM_ALLOC);
if (!area)
    return NULL;
```

Calculamos el número el número de páginas y el total de bytes que solicitamos

```
nr_pages = size >> PAGE_SHIFT;
array_size = (nr_pages * sizeof(struct page *));
area->nr_pages = nr_pages;
```

Reservamos espacio en función de nuestro array_size. Si este es mayor que el tamaño de una página entonces llamamos a la función vmalloc que nos permite reservar páginas no contiguas, sino llamamos a kmalloc.

```
if (array_size > PAGE_SIZE)
        pages = __vmalloc(array_size, gfp_mask, PAGE_KERNEL);
else
        pages = kmalloc(array_size, (gfp_mask & ~__GFP_HIGHMEM));
area->pages = pages;
```

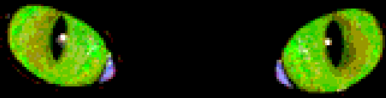
Si no encontramos espacio suficiente eliminamos todo el espacio reservado y finalizamos devolviendo NULL.

```
if (!area->pages) {
        remove_vm_area(area->addr);
        kfree(area);
        return NULL;}
```

Utilizando la función memset ponemos a nulo todo el espacio de memoria que indica la variable array_size en la dirección que indica area->pages.

```
memset(area->pages, 0, array_size);
```

Llamada VMALLOC (III)



Con la función alloc_page enmarcamos cada página y almacenamos el correspondiente descriptor de cada una de las páginas en el vm_struct. Por último, actualizamos el número de páginas totales.

```
for (i = 0; i < area->nr_pages; i++) {
    area->pages[i] =
    alloc_page(gfp_mask);
        if (unlikely(!area->pages[i])) {
            area->nr_pages = i;
            goto fail;
```

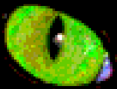
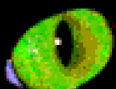
Mediante el map_vm_area mapeamos a direcciones no contiguas de memoria física, el espacio virtual de direccionamiento utilizado por el núcleo (contiguo). Si no podemos ubicar las páginas en la zona de memoria devolvemos un fallo. Sino se devolverá la dirección donde reside nuestra zona de memoria

```
if (map_vm_area(area, prot, &pages))
    goto fail;
return area->addr;
fail:
    vfree(area->addr);
    return NULL;}
```

LLamada GET_VM_AREA

- Esta función se encarga de buscar un hueco libre en la zona de memoria del kernel, del tamaño definido en el parámetro de la función
- El llamador es el responsable de que el tamaño solicitado sea distinto de cero y múltiplo del tamaño de página.
- Habitualmente utiliza la política first-fit algoritmo para localizar la zona de memoria, aunque podría modificarse la política de selección del espacio de memoria siguiendo otras políticas de búsqueda.

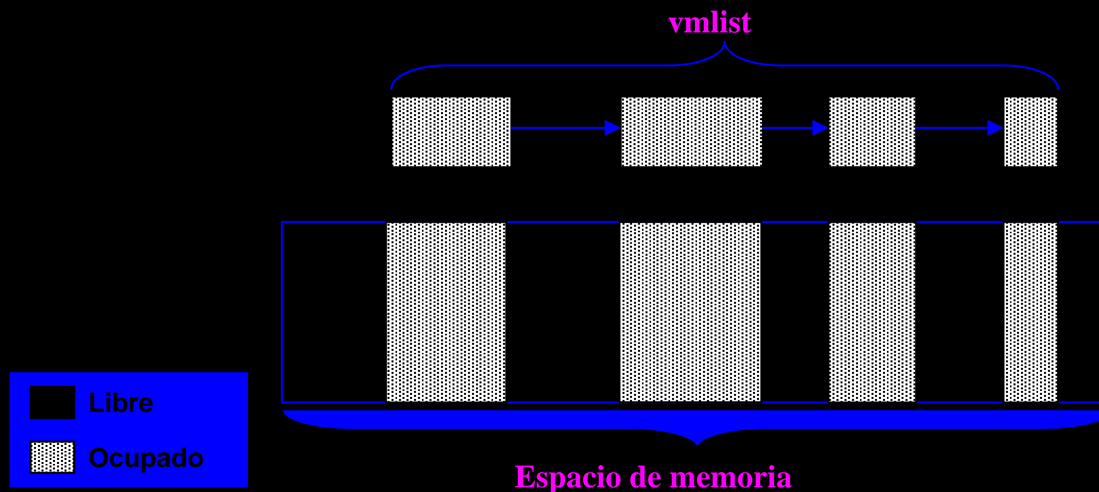
```
vm_struct * get_vm_area (unsigned long size)
```

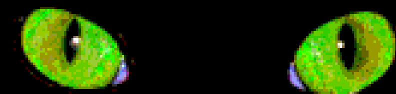



Índice

LLamada GET_VM_AREA (I)

- Utiliza la variable global vmlist que contiene una lista ordenada de trozos de memoria ocupada.





LLamada GET_VM_AREA (II)

- **PASOS DEL ALGORITMO**

Esta función actúa con dos parámetros: el tamaño en bytes de la región de memoria creada y un flag que especifica el tipo de región.

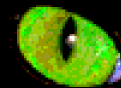
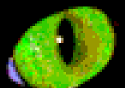
```
struct vm_struct *__get_vm_area(unsigned long size, unsigned long flags,
                                unsigned long start, unsigned long end)
```

1. Kmalloc obtiene un area de memoria para el nuevo descriptor vm_struct.malloc( )

```
addr = ALIGN(start, align);
area = kmalloc(sizeof(*area), GFP_KERNEL);
```

Si no encontramos el hueco libre, liberamos la memoria reservada para el nuevo elemento de vmlist

```
if (unlikely(!area))
        return NULL;
        size += PAGE_SIZE;
        if (unlikely(!size)) {
                kfree (area);
                return NULL;}
```

LLamada GET_VM_AREA (III)

2. Utiliza la vmlist_lock para bloquear la escritura y a continuación escanea la lista de descriptores a fin de buscar un hueco libre.

```
write_lock(&vmlist_lock);
for (p = &vmlist; (tmp = *p) != NULL ;p = &tmp->next) {
        if ((unsigned long)tmp->addr < addr) {
                if((unsigned long)tmp->addr + tmp->size >= addr)
                        addr = ALIGN(tmp->size + (unsigned long)tmp->addr, align);
        }
        if ((size + addr) < addr) goto out;
                if (size + addr <= (unsigned long)tmp->addr)
                        goto found;
                addr = ALIGN(tmp->size + (unsigned long)tmp->addr, align);
                if (addr > end - size) goto out;
```

3. Si tal intervalo existe, la función inicializa los campos del vm_struct, libera el bloqueo de escritura y termina. De otro modo, get_vm_area( ) libera el descriptor obtenido previamente, libera el bloqueo de escritura y retorna NULL.

```
found:
area->next = *p;
*p = area;
area->flags = flags;
area->addr = (void *)addr;
```

```
out:
write_unlock(&vmlist_lock);
kfree(area);
return NULL;
```

La función vfree( ) libera áreas de memoria no contiguas creadas por vmalloc( ), mientras que la función vunmap( ) libera memoria de áreas creadas por vmap( ). Ambas funciones tiene de parámetro la dirección inicial del área que va a ser liberado.

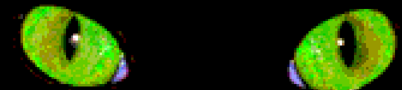
```
void vfree(void *addr)
{
    BUG_ON(in_interrupt());
    __vunmap(addr, 1);
}
```

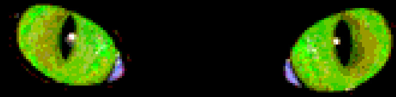
Índice

LLamada VFREE

La función vunmap( ) recibe dos parámetros: la dirección inicial del área que va a ser liberado y el flag deallocate_pages, que se activa si la página mapeada en el área debería ser liberada.

```
Void __vunmap(void *addr, int Deallocate_pages)
{
        struct vm_struct *area;
        if (!addr)
        return;
...
```

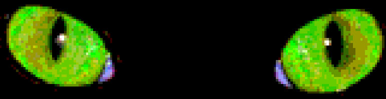




LLamada VFREE

Dirección incorrecta. No es múltiplo del tamaño de página (no está alineada)

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



PASOS DEL ALGORITMO

1. Invoca a **remove_vm_area( )** para conseguir la dirección del área apuntada por el descriptor vm_struct y limpiar las entradas de la tabla de páginas del núcleo correspondientes a las direcciones en el área de memoria no contigua.

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

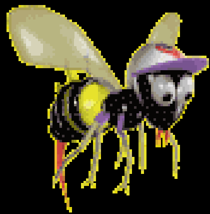
PASOS DEL ALGORITMO

2 **Si flag deallocate_pages activo, este escanea el array area->pages y para cada elemento invoca free_page( ) que se encarga de liberar cada página. Además ejecuta kfree para liberar dicho array.**

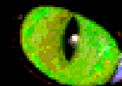
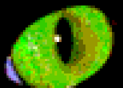
```
if (deallocate_pages) {
        int i;
        for (i = 0; i < area->nr_pages; i++) {
                if (unlikely(!area->pages[i]))
                BUG();
                __free_page(area->pages[i]);
        }
        if (area->nr_pages > PAGE_SIZE/sizeof(struct
        page *))
                vfree(area->pages);
        else
                kfree(area->pages);
}
```

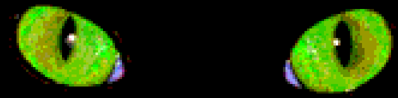
PASOS DEL ALGORITMO

3. Invoca kfree(area) para liberar el descriptor de vm_struct.



```
    kfree(area);
    return;
}
...
```

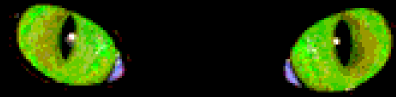


Mprotect

- Se utiliza para modificar los derechos de acceso asociados a una parte de la memoria. mprotect controla la forma en que una sección de memoria puede ser accedida.
- El prototipado de la función es:

```
sys_mprotect(unsigned long start, size_t len,
    unsigned long prot)
```

Índice



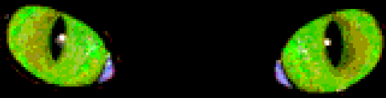
Mprotect

- Donde:

Start: es la dirección de la memoria sobre la que queremos cambiar el acceso

Len: longitud de la zona de memoria en bytes

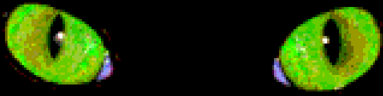
Prot: es el tipo de protección que queremos establecer, que puede tomar 6 valores (PROT_NONE, PROT_READ, PROT_WRITE, PROT_EXEC, PROT_SEM, PROT_GROWSDOWN, PROT_GROWSUP):



Mprotect

Estos valores corresponden a macros definidas en mman.h

- **PROT_NONE**: La memoria no se puede acceder
- **PROT_READ**: Solo hay permiso de lectura
- **PROT_WRITE**: Solo hay permiso de escritura
- **PROT_EXEC**: Puede contener código ejecutable
- **PROT_SEM**: Suede ser usada para operaciones atómicas
- **PROT_GROWSDOWN**: marca para inicio de área de memoria virtual que está decreciendo.
- **PROT_GROWSUP**: marca para fin de área de memoria virtual que está creciendo.



Mprotect

- La función devuelve la variable error que puede tomar los valores:

EINVAL: parámetros no válidos

EFAULT: la memoria no puede ser accedida

ENOMEM: no hay memoria libre

EACCES: la memoria no permite el acceso especificado, por ejemplo, si se asocia a una zona de memoria mediante mmap, un fichero que solo tiene acceso de lectura y se le pide a mprotect que lo marque con PROT_WRITE

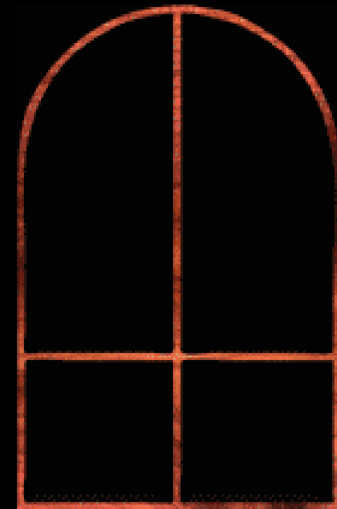
PASOS DEL ALGORITMO

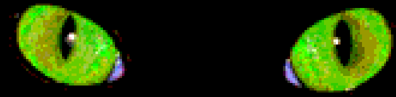
1. Comprobamos si las protecciones de la zona de memoria Especificada por los parámetros de entrada (comienzo, final y permisos) son correctos y además en que situación nos encontramos para a continuación modificar los permisos que indica el parámetro ***prot*.**

2. Si hasta este punto todo el chequeo ha sido correcto entonces ya podemos modificar las protecciones para dicha zona de memoria pero antes hay que comprobar que dichas modificaciones se pueden aplicar.

3. Por último las nuevas protecciones se establecen mediante la función ***mprotect_fixup*** la cuál llama a una función según la ubicación de la zona a modificar en la región de memoria. Seguidamente llama a ***change_protection*** para modificar las protecciones asociadas a las páginas de memoria correspondientes

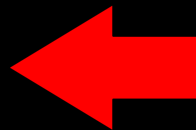
FIN

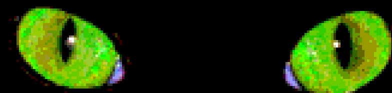




Heap

- Es la zona que queda disponible para la asignación de memoria dinámica
- Su tamaño máximo depende del tamaño máximo de la pila y viceversa ver dibujo





void main(void)

[...]

char * p_nombre

(espacio para 8 chars)

PILA

HEAP

Otras Regiones

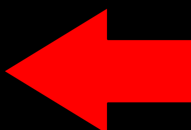
crecimiento de la pila

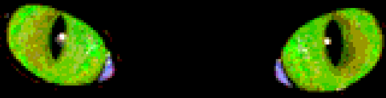
crecimiento del heap

0xBFFFFFFF

total de memoria disponible para un proceso

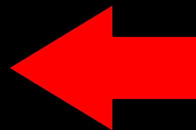
0x00000000

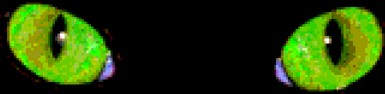




Stack

- Cada vez que se llama a una función se introduce en el Stack la información necesaria para poder retomar la ejecución de esta, como son los parámetros pasados y las variables declaradas en ella.

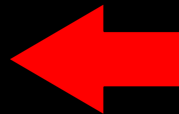


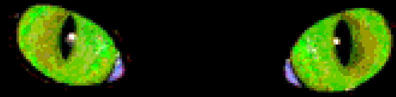


Malloc

- Solicita al sistema operativo memoria dinámica
- Devuelve un puntero que apunta a la zona de memoria asignada o nill en caso de no poderse atender la solicitud.
- La forma de llamarla es:

void *malloc(size_t size);

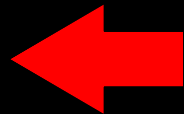


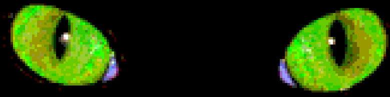


Calloc

- Asigna espacio para una array de n objetos y con tamaño de objeto tam.
- Devuelve este espacio inicializado a cero todos los bits
- La forma de llamarlo es:

```
void *calloc(size_t n, size_t tam);
```

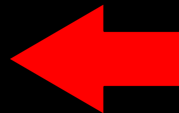




Realloc

- Cambia el tamaño de una zona de memoria que ya ha sido asignada con malloc
- La forma de llamarla es:

```
void *realloc(void *ptr, size_t size);
```



- Libera una zona memoria que había sido asignada con malloc
- La forma de llamarla es

void free(void *ptr);