



# **Sistemas Operativos**

## **Memoria II**

---

# Administración de Memoria Kernel

## Conceptos clave:

- Mapeo directo
- kalloc
- Configuración de páginas



## Mapeo directo en xv6

El kernel accede a la RAM y a los registros de los dispositivos mapeados en memoria utilizando "mapeo directo"; es decir, asignando los recursos a **direcciones virtuales que son iguales a las direcciones físicas**.

Por ejemplo, el propio kernel está ubicado en `KERNBASE=0x80000000` tanto en el espacio de direcciones virtuales como en la memoria física.



## Mapeo directo en xv6

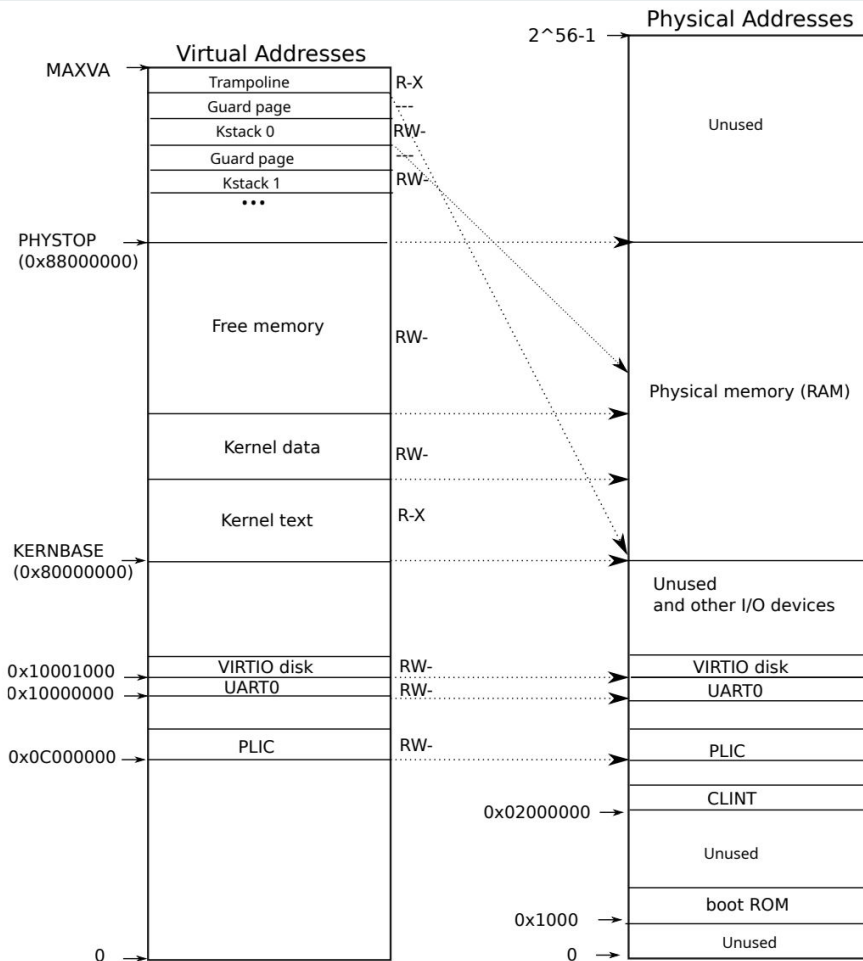
El mapeo directo simplifica el código del kernel que lee o escribe en la memoria física.

Por ejemplo, cuando fork asigna memoria de usuario para el proceso hijo, el asignador devuelve la dirección física de esa memoria; fork utiliza esa dirección directamente como una dirección virtual cuando copia la memoria de usuario del padre al hijo.

# Mapeo directo

Hay un par de direcciones virtuales del kernel que no están mapeadas directamente:

- **La página trampolín.** Está mapeada en la parte superior del espacio de direcciones virtuales tanto para el espacio de kernel como usuario.
- **Las páginas de pila del kernel.** Cada proceso tiene su propia pila del kernel, que está mapeada en una dirección alta para que debajo de ella xv6 pueda dejar una página de protección sin mapear.





## Kalloc: alocador de memoria fisica

- La estructura de datos del asignador es una lista libre de páginas de memoria física que están disponibles para asignación.
- Cada elemento de la lista de páginas libres es un struct run.
- ¿De dónde obtiene el asignador la memoria para almacenar esa estructura de datos? Almacena la estructura run de cada página libre en la propia página libre, ya que no hay nada más almacenado allí.

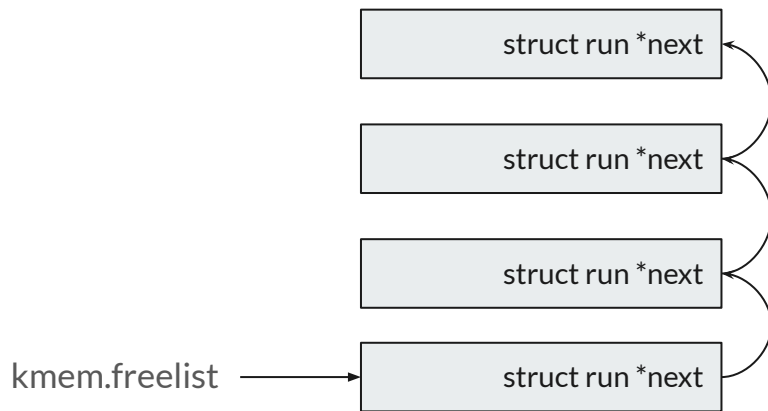
```
struct run {  
    struct run *next;  
};
```

## kalloc: alocador de memoria fisica

```
void * kalloc(void){
    struct run *r;

    acquire(&kmem.lock);
    r = kmem.freelist;
    if(r)
        kmem.freelist = r->next;
    release(&kmem.lock);

    if(r)
        memset((char*)r, 5, PGSIZE);
    return (void*)r;
}
```



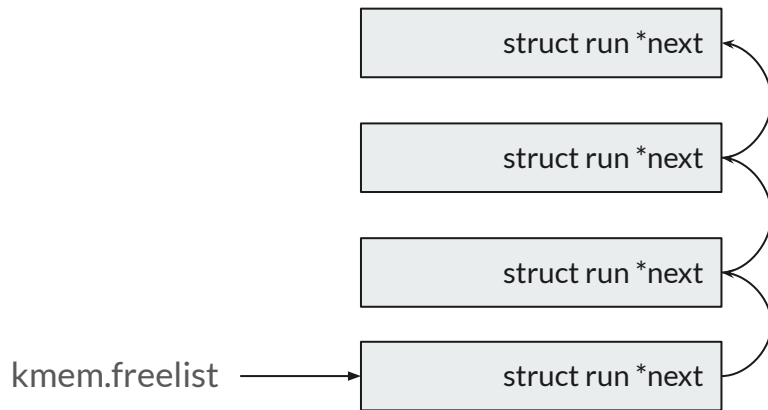
# kfree

```
void kfree(void *pa) {
    struct run *r;

    ...

    r = (struct run*)pa;

    acquire(&kmem.lock);
    r->next = kmem.freelist;
    kmem.freelist = r;
    release(&kmem.lock);
}
```







## **kalloc: alocador de memoria fisica**

Notar que:

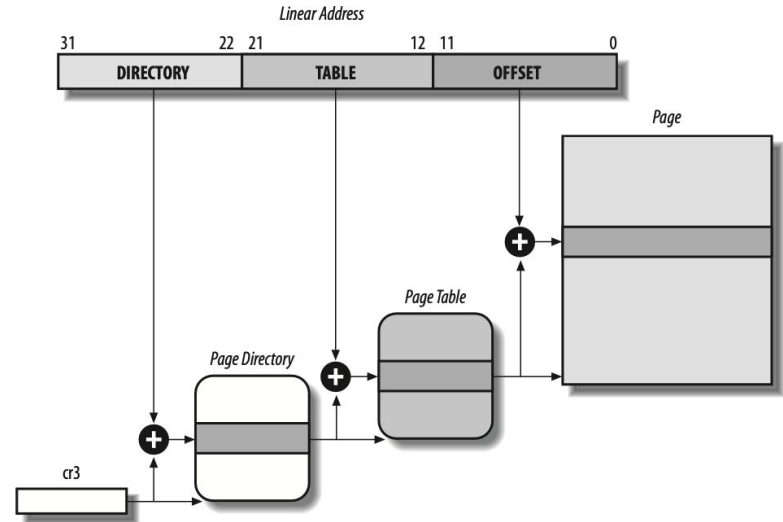
- kalloc no recibe parámetros porque siempre aloca de a una página y la pagina siempre tiene tamaño fijo (eg. 4 Kb)
- Tanto kalloc como kfree operan con direcciones físicas, no virtuales. Otra función se encarga de mapearlas en las tablas de páginas
- La lista en realidad funciona como una pila
- Para que no se corrompa la estructura de la pila, la modificación se realiza en una región crítica, que se construye con locks (ver clase sobre concurrencia)

# Mapeo de páginas

El objetivo es dada la dirección virtual de una página, alocar una pagina física y mapearla a esa direccion virtual. Esto es lo que hace el kernel cada vez que un proceso pide más memoria.

Ejemplo con un esquema de 2 niveles en x86 (no xv6):

Recordar que los bits 31-22 y 21-12 forman **indices** dentro de las tablas de páginas correspondientes





## Mapeo de páginas

El objetivo es dada la dirección virtual de una página, alocar una pagina fisica y mapearla a esa dirección virtual. Esto es lo que hace el kernel cada vez que un proceso pide más memoria.

```
int mappage(pte_t *pagetable, uint32 va)
```

- `pte_t *pagetable` representa un array de `pte_t`, es decir una tabla de páginas. `pte_t` es un tipo de dato abstracto que representa una page table entry.
- `uint32 va` es la dirección de la página virtual que se desea mapear. Asumir que no se encuentra ya mapeada



## Mapeo de páginas

Asuma que dispone de las siguientes funciones (en verde en el código siguiente):

- `void* kalloc()` : aloca y devuelve la dirección física de un frame de memoria disponible

Tipo de dato abstracto `pte_t`

- `void pte_init(pte_t* pte, uint32 pa)` : inicializa la pte para apuntar a la dirección física pa
- `uint32 pte_pa(pte_t* pte)` : devuelve la dirección física asociada al pte, o 0 si la pte no está mapeada.

*En este ejemplo vamos a ignorar los permisos y flags de las páginas*

# Mapecto de págimas

```
1 #define PAGE_SIZE 4096           // Tamaño de la página: 4KB
2 #define PTE_COUNT 1024           // Número de entradas por tabla (2^10)
3
4 typedef uint32 pte_t;             // Definimos pte_t como un entero de 32 bits
5
6
7 int mappage(pte_t *pagetable, uint32 va) {
8
9
10     // Máscaras para extraer los índices de las tablas de página
11     uint32 idx_l1 = (va >> 22) & 0x3FF;    // Los primeros 10 bits
12     uint32 idx_l2 = (va >> 12) & 0x3FF;    // Los siguientes 10 bits
13
14     ...
15 }
```

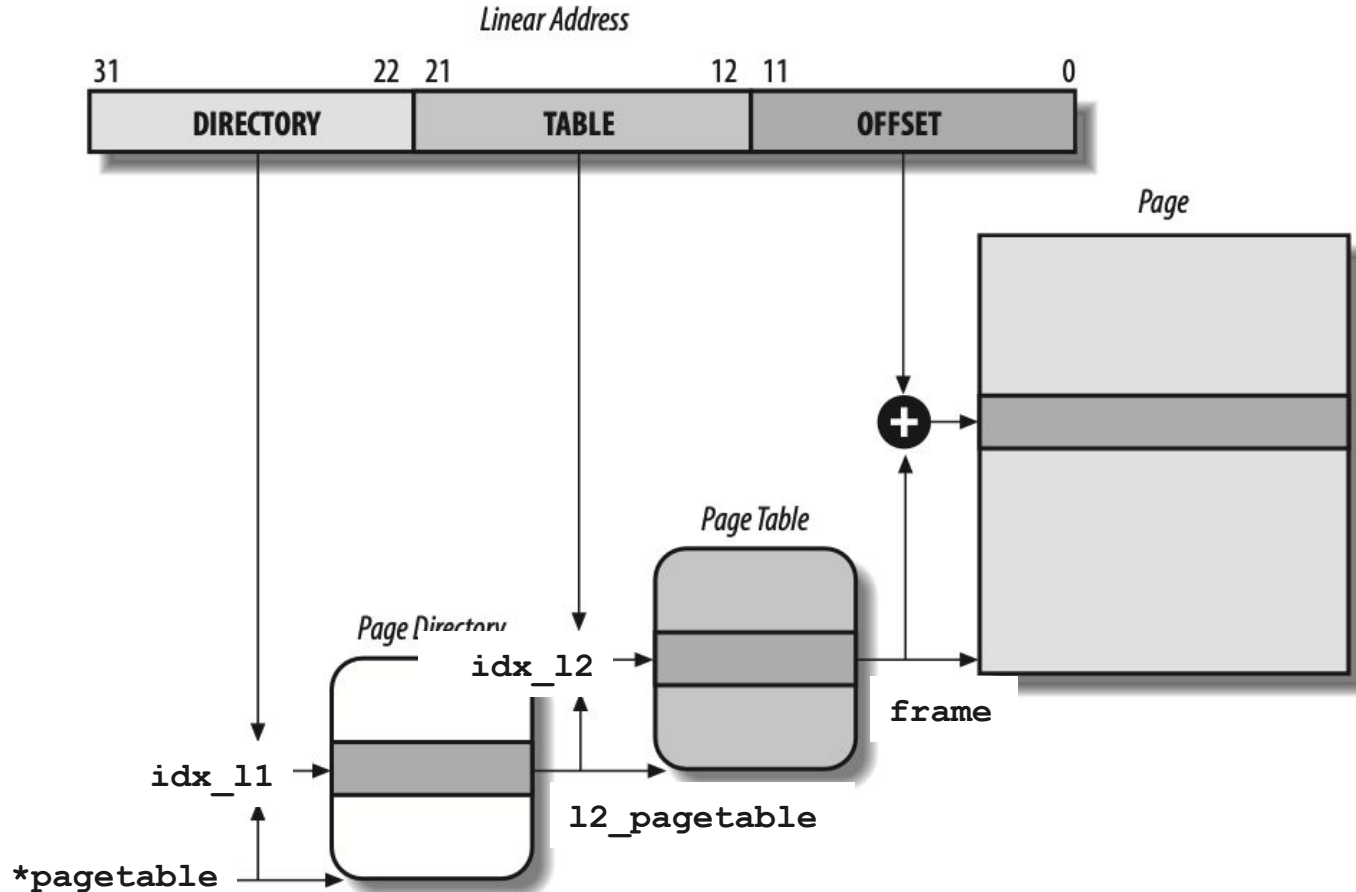
# Maapeo de páginas

```
...
14 // Obtenemos la entrada de la tabla de primer nivel (nivel 1)
15 pte_t *l2_pagetable = (pte_t *) pte_pa(&pagetable[idx_l1]);
16
17 // Si la tabla de nivel 2 no existe, la creamos
18 if (l2_pagetable == 0) {
19
20     // Asignamos un nuevo frame de memoria física para la tabla de segundo nivel
21     l2_pagetable = (pte_t *)kalloc();
22
23     if (l2_pagetable == 0) {
24         // Error: No se pudo asignar memoria
25         return -1;
26     }
27
28     // Inicializamos la entrada de la tabla de nivel 1
29     // para apuntar a la nueva tabla de nivel 2
30     pte_init(&pagetable[idx_l1], (uint32)l2_pagetable);
31 }
32
...
```

# Mapeo de páginas

```
...
33 // Asignamos un nuevo frame de memoria física para la página
34 void *frame = kalloc();
35 if (frame == 0) {
36     // Error: No se pudo asignar memoria
37     return -1;
38 }
39
40 // Inicializamos la entrada de la tabla de nivel 2
41 // para apuntar al frame de memoria física
42 pte_init(&l2_pagetable[idx_l2], (uint32)frame);
43
44 // Éxito
45 return 0;
46 }
```

# Mapeo de páginas: resumen de punteros





# Páginas en xv6 RISC-V

- Las páginas en RISC-V de 64 bits tienen 3 niveles en lugar de 2.
- Los 25 bits superiores de la dirección virtual se ignoran
- El bit V - Valid indica si la página apunta a otra página válida o si no está mapeada

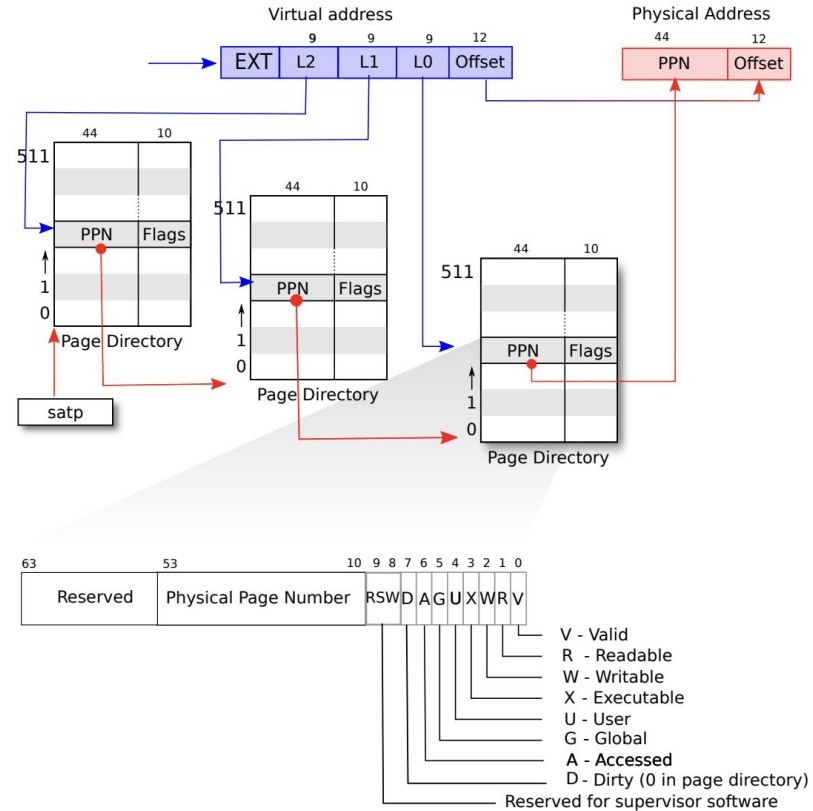


Figure 3.2: RISC-V address translation details.

# Función walk en xv6

Se usa para obtener la pte\_t de nivel inferior dada una virtual address

```
pte_t * walk(pagetable_t pagetable, uint64 va, int alloc)
{
    if(va >= MAXVA)
        panic("walk");

    for(int level = 2; level > 0; level--) {
        pte_t *pte = &pagetable[PX(level, va)];

        if(*pte & PTE_V) { // Si la pagina esta presente
            pagetable = (pagetable_t)PTE2PA(*pte);
        } else {
            if(!alloc || (pagetable = (pde_t*)kalloc()) == 0)
                return 0;
            memset(pagetable, 0, PGSIZE);
            *pte = PA2PTE(pagetable) | PTE_V;
        }
    }
    return &pagetable[PX(0, va)];
}
```

---

# Administración de Memoria Usuario

## Conceptos clave:

- malloc
- mmap



## malloc()

```
#include <stdlib.h>
```

```
void *malloc(size_t size);
```

Returns: ptr to allocated block if OK, NULL on error

- En unix malloc() devuelve un bloque de size bytes alineado a 8-bytes (double word).
- No inicializa la memoria devuelta.
- Utiliza la system call sbrk o mmap.



## free()

```
#include <stdlib.h>

void free(void *ptr);
```

Returns: nothing

Libera bloques reservados en el heap.

El ptr debe haber sido reservado previamente con malloc(), calloc() o realloc(). Si esto no sucede el comportamiento de free es INDEFINIDO

# Ejemplo

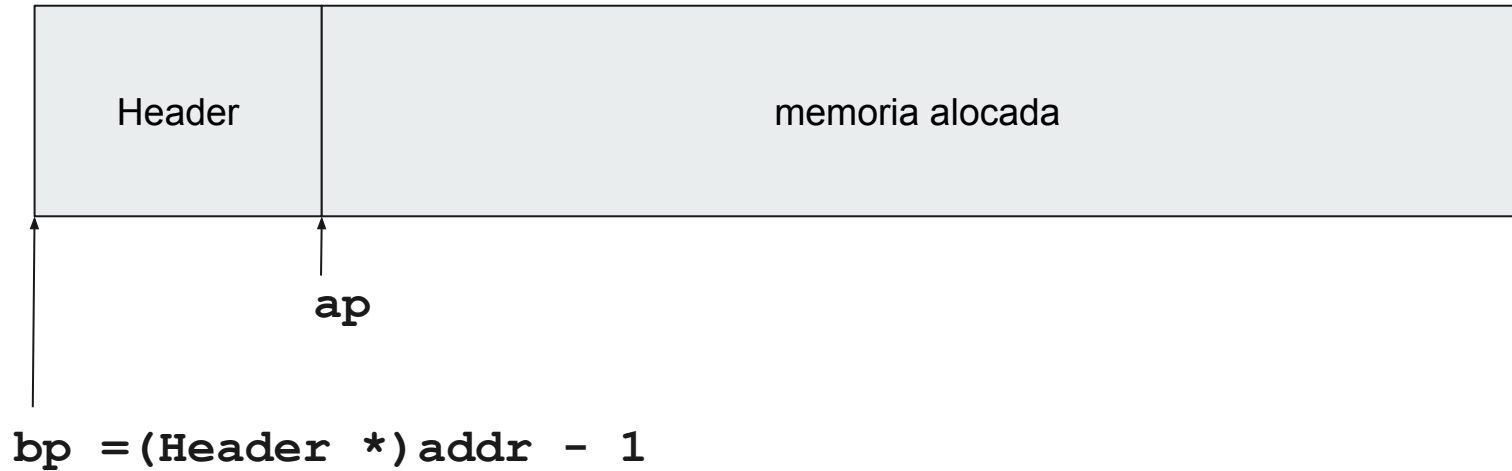
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p1 = malloc(4*sizeof(int)); // allocates enough for an array of 4 int
    int *p2 = malloc(sizeof(int[4])); // same, naming the type directly
    int *p3 = malloc(4*sizeof *p3); // same, without repeating the type name

    if(p1) {
        for(int n=0; n<4; ++n) // populate the array
            p1[n] = n*n;
        for(int n=0; n<4; ++n) // print it back out
            printf("p1[%d] == %d\n", n, p1[n]);
    }

    free(p1);
    free(p2);
    free(p3);
}
```

`ap = malloc(n)`



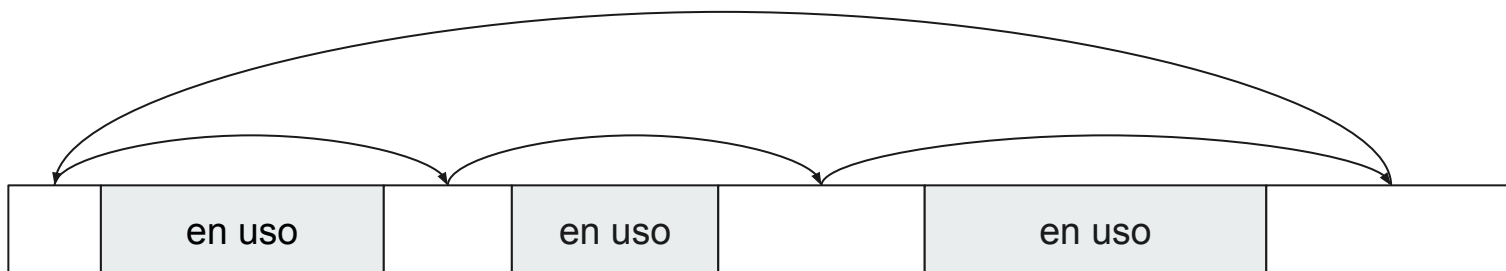


## Estructura Header

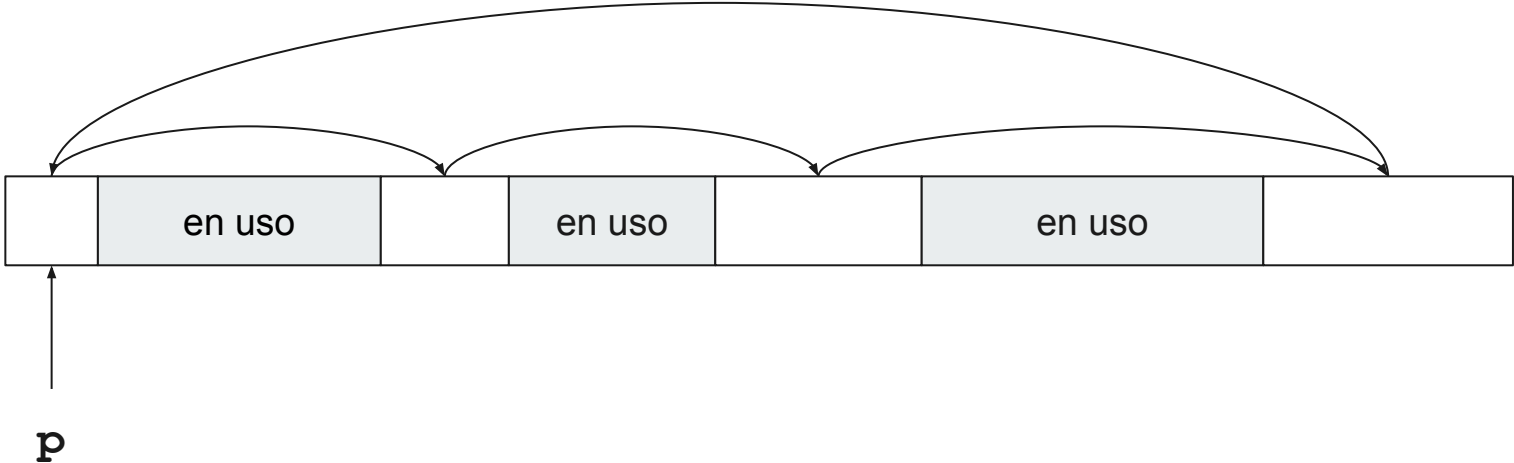
```
struct header {  
    struct header *ptr;  
    unsigned int size;  
};
```

```
typedef struct header Header;
```

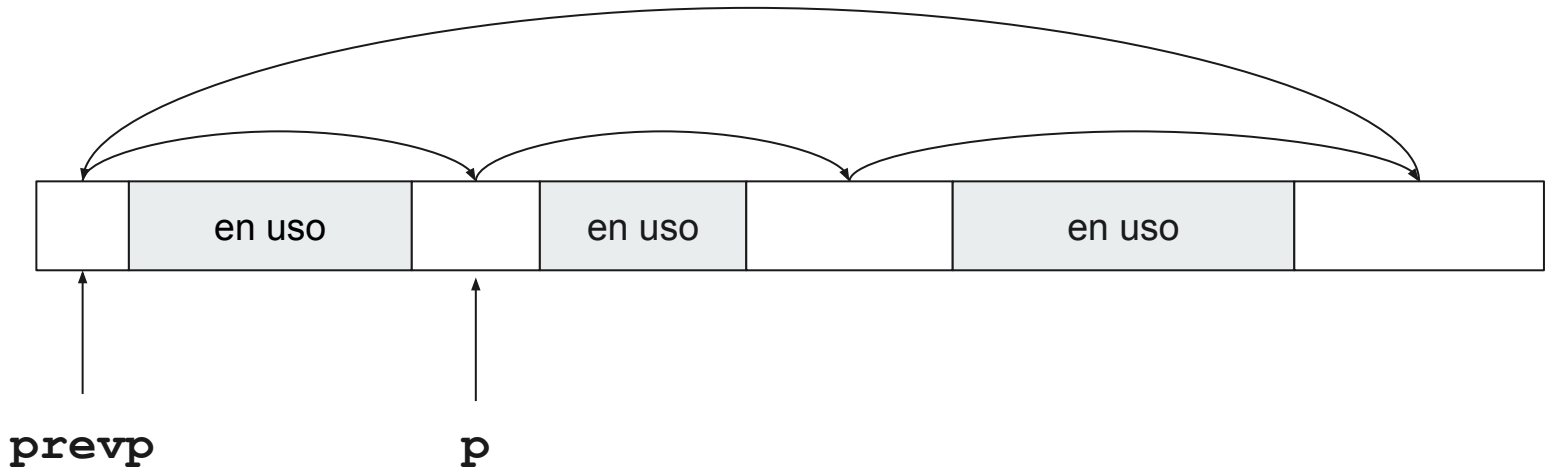




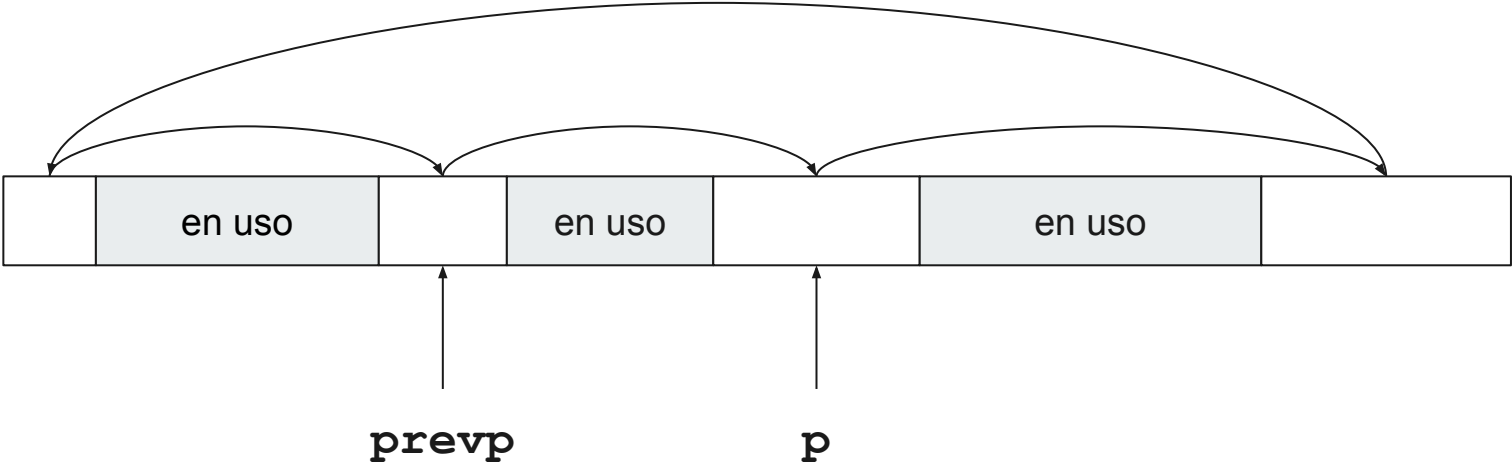
# Buscar un bloque libre



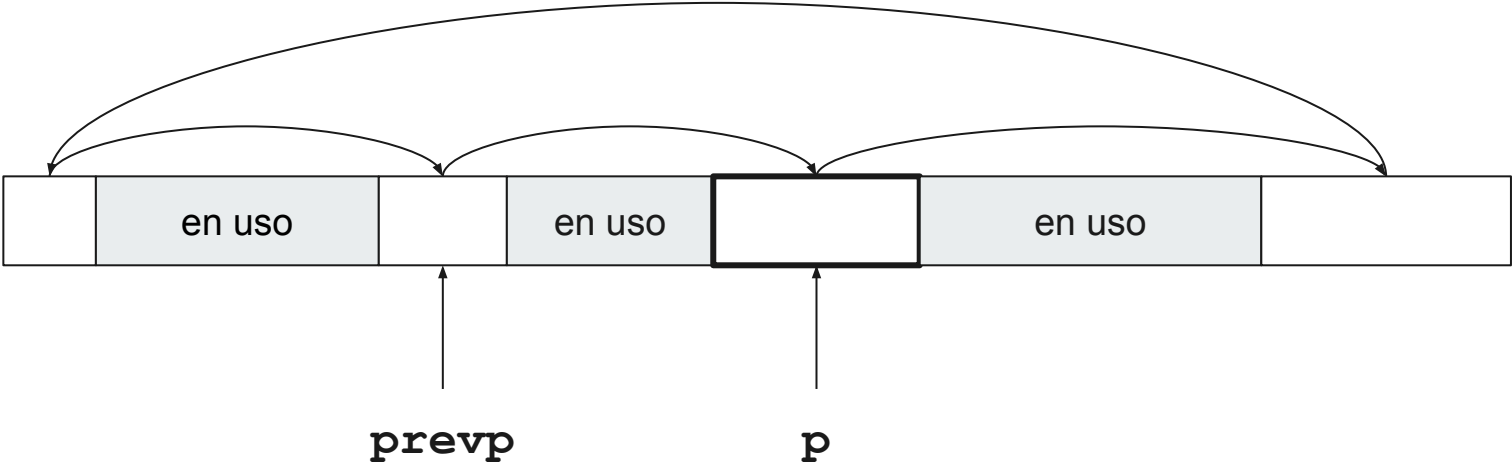
# Buscar un bloque libre



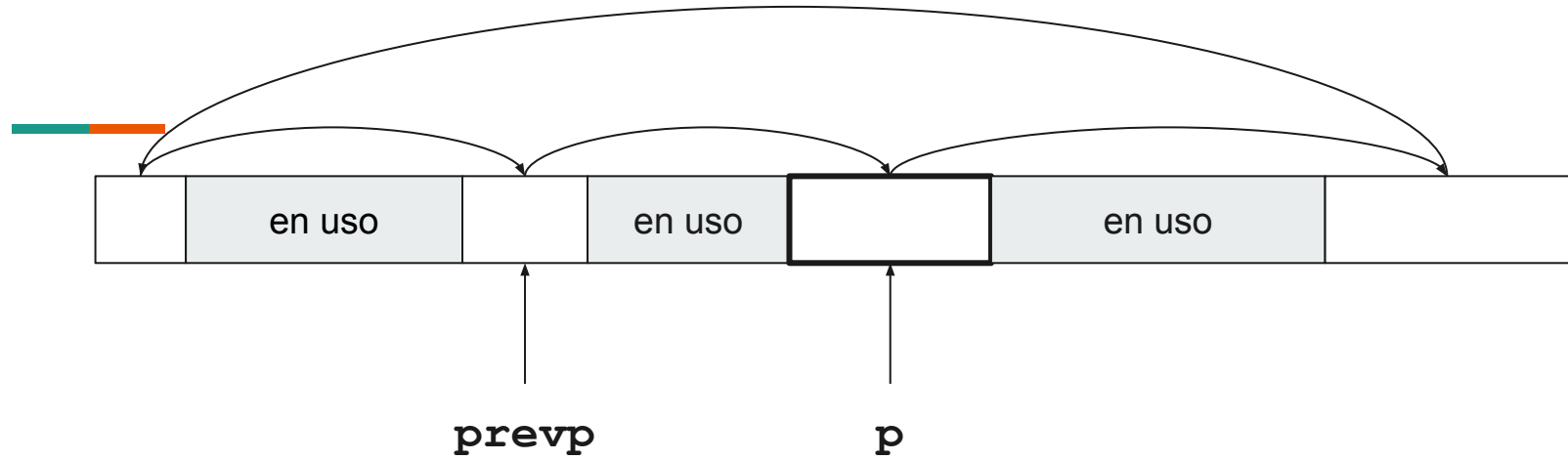
# Buscar un bloque libre



# Buscar un bloque libre

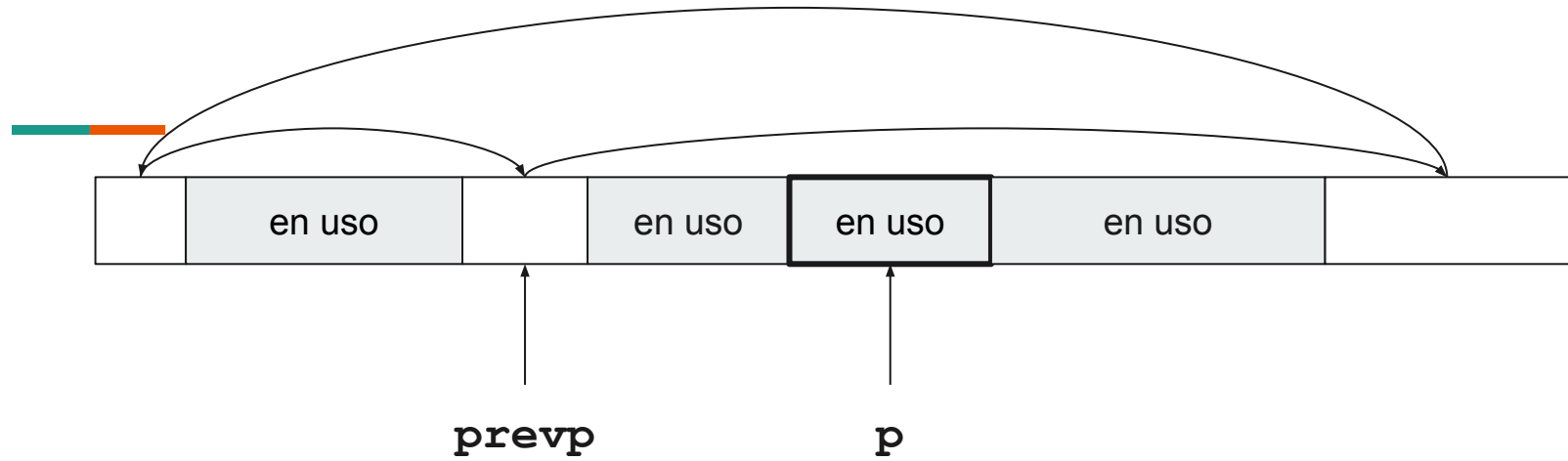


## Opcion 1: Reservar el bloque completo



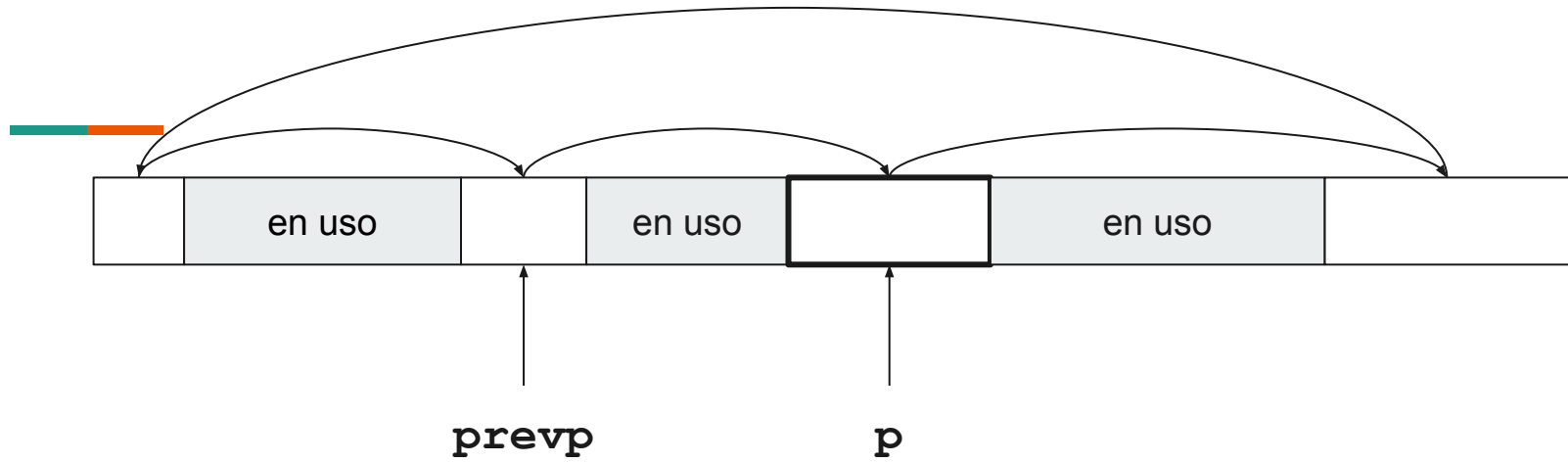
```
prevp->ptr = p->ptr;  
return (p+1); // p es de tipo Header
```

## Opcion 1: Reservar el bloque completo



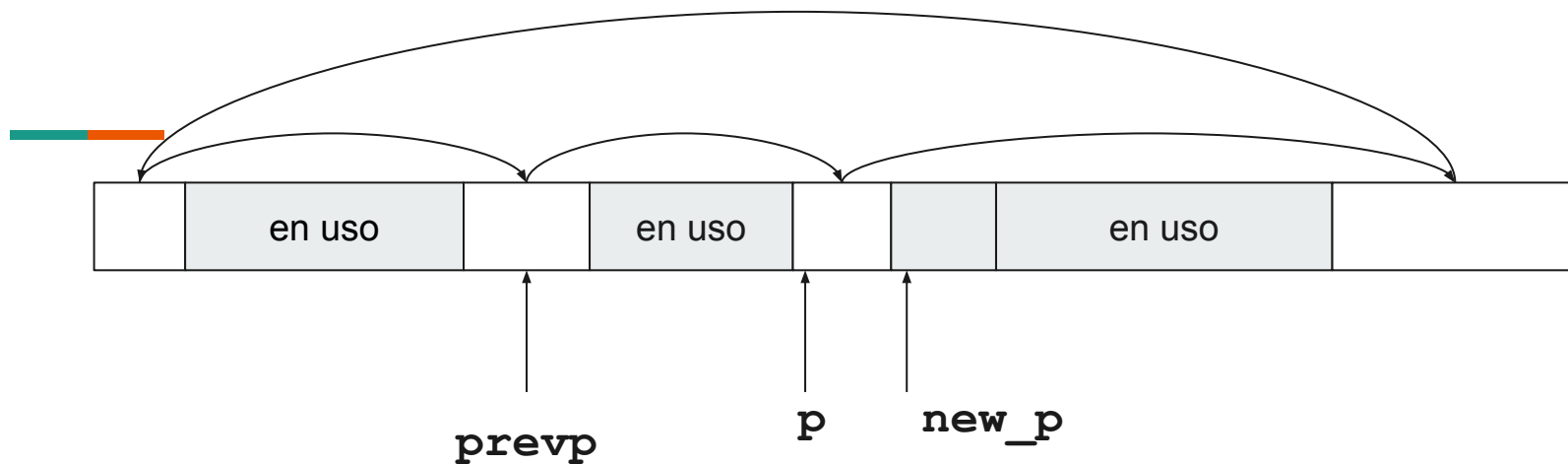
```
prevp->ptr = p->ptr;  
return (p+1); // p es de tipo Header
```

## Opcion 2: Split





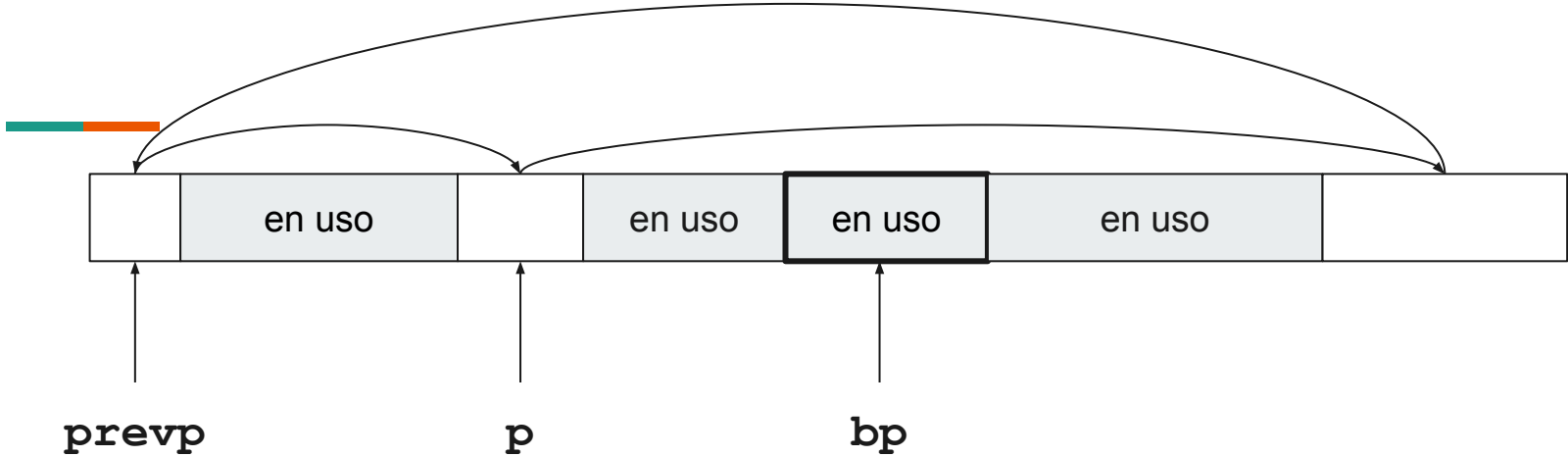
## Opcion 2: Reservar parte del bloque



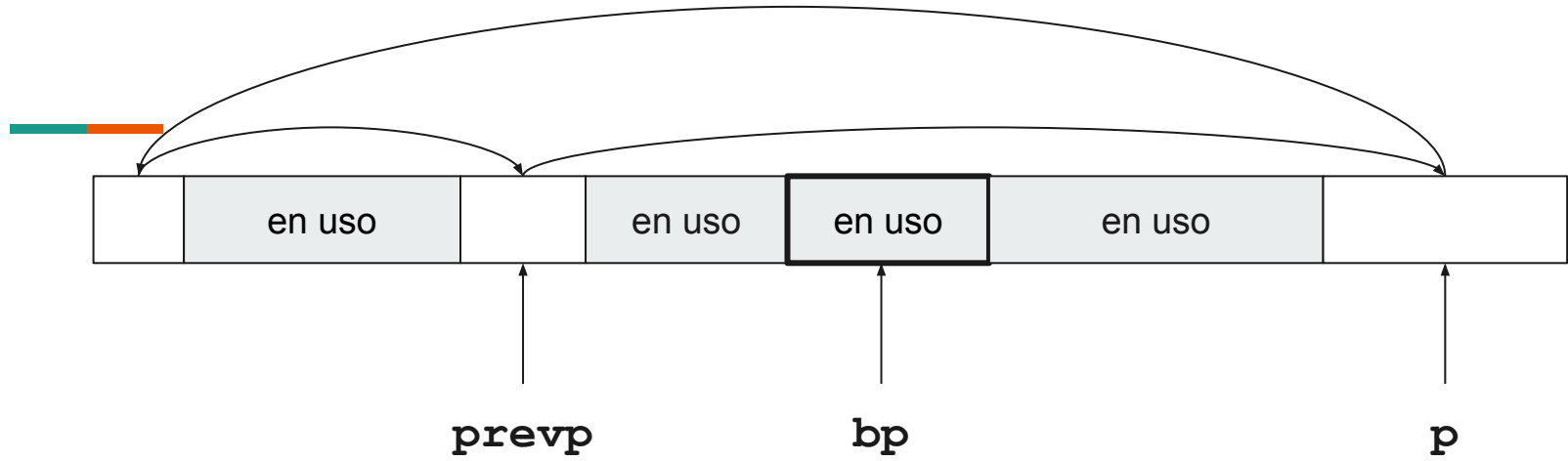
```
p->size -= nunits;  
new_p = p + p->size;  
new_p->size = nunits;
```

```
return (new_p + 1); // new_p es de tipo Header
```

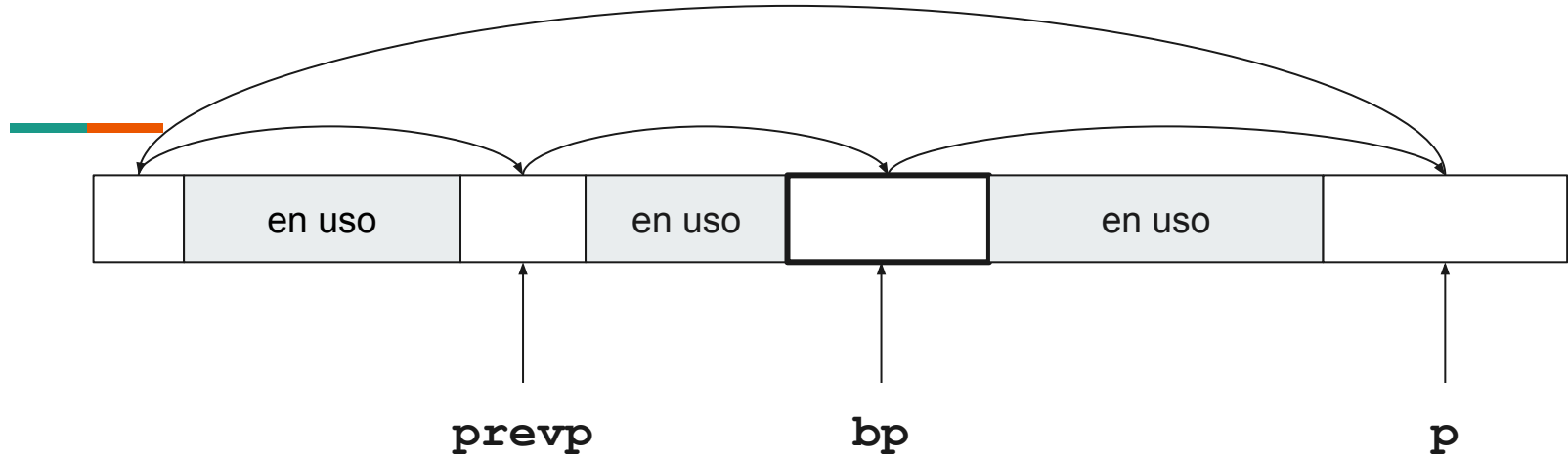
# Liberar el bloque



# Liberar el bloque

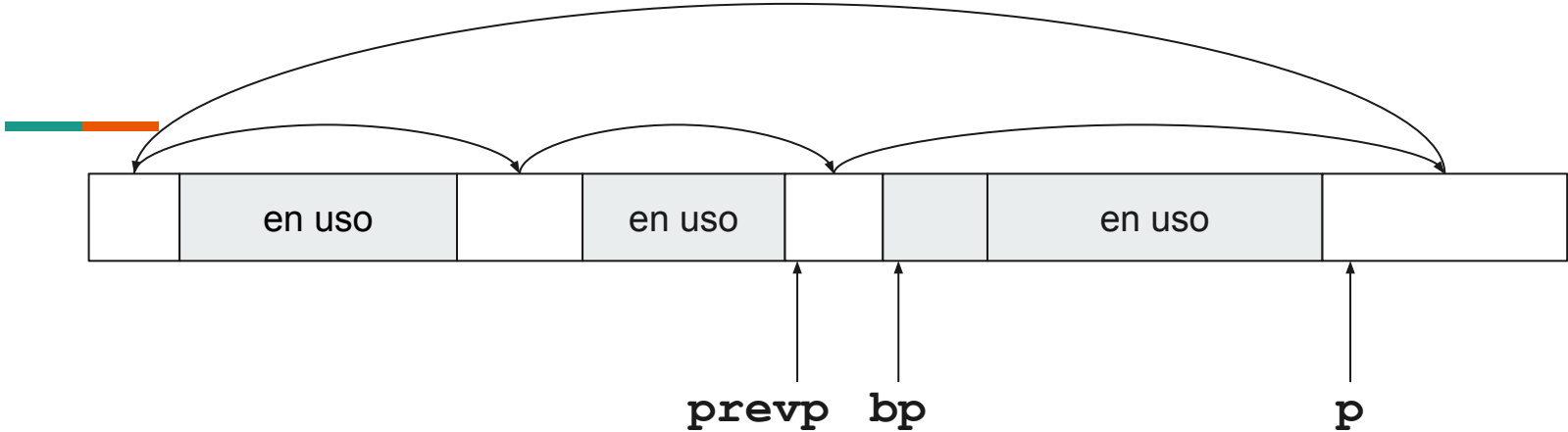


# Liberar el bloque

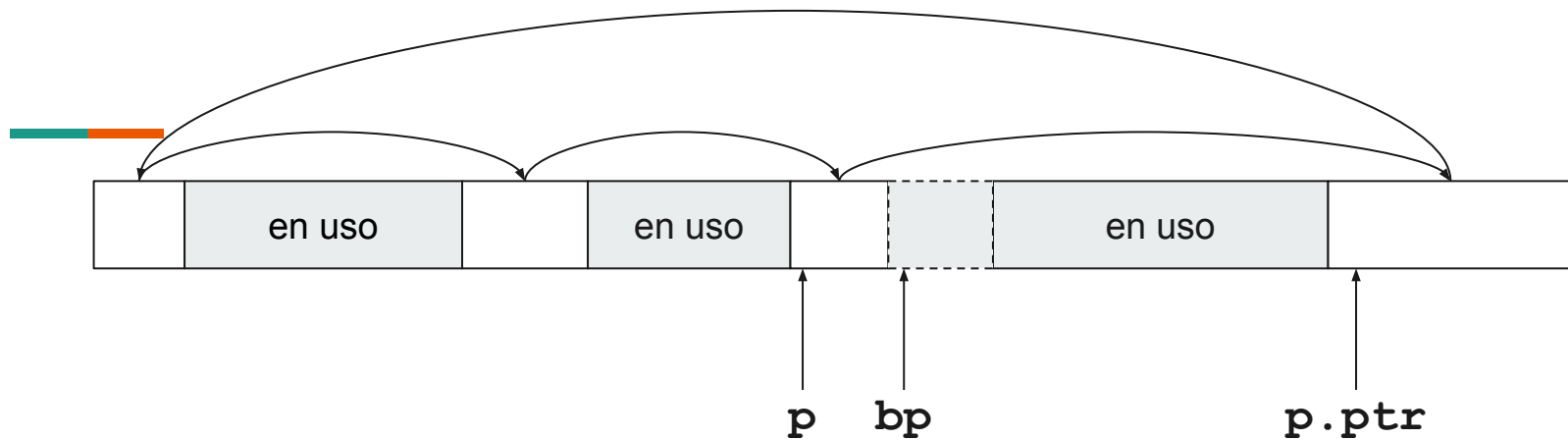


```
bp->ptr = p  
prevp->ptr = bp;
```

# Liberar bloque: Coalesce

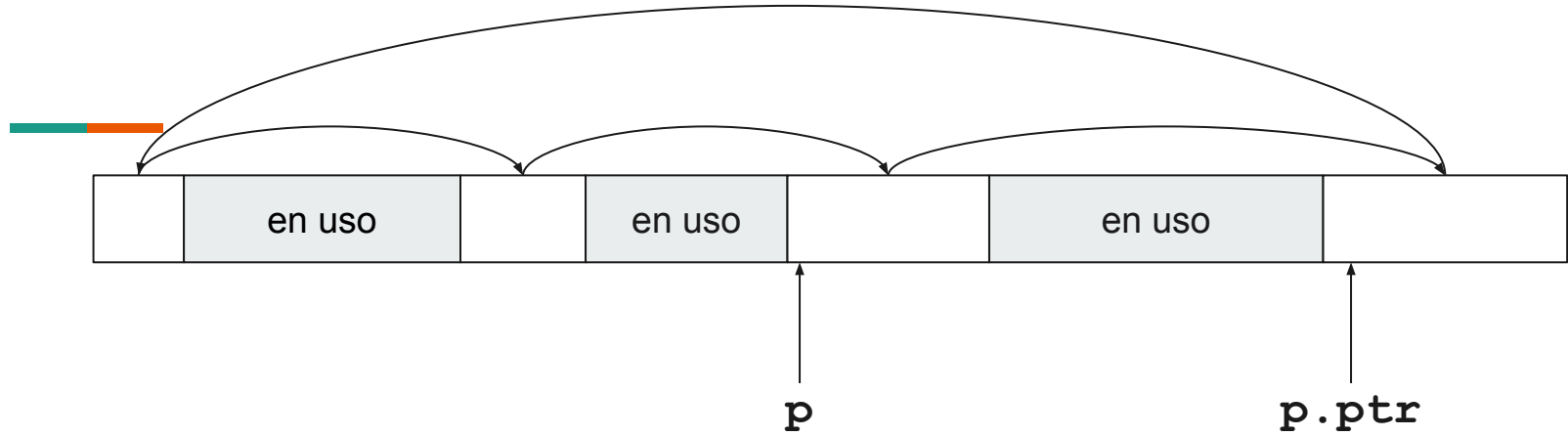


## Liberar bloque: Coalesce



```
if (p + p->size == bp) {  
    p->size += bp->size  
}
```

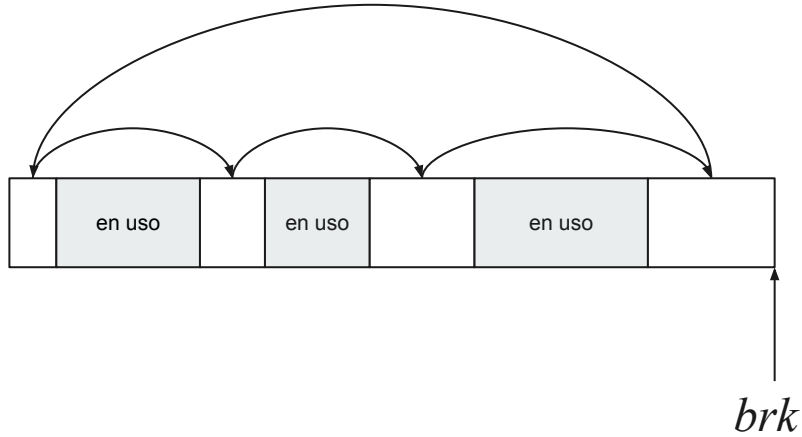
## Liberar bloque: Coalesce



```
if (p + p->size == bp) {  
    p->size += bp->size  
}
```

# Reservar memoria pidiendo mas al OS

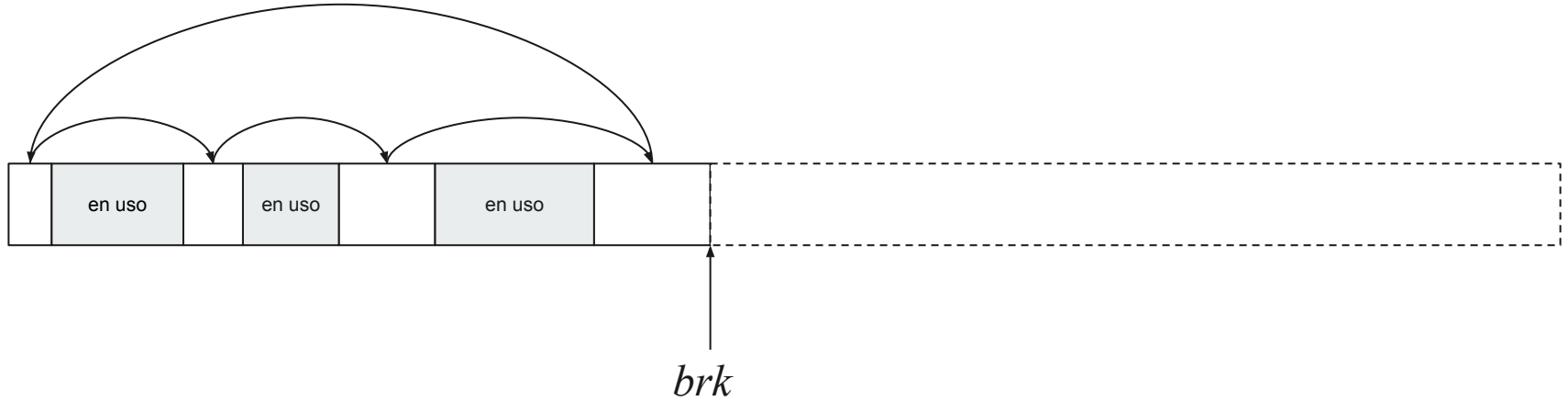
```
ap = malloc(n)
```





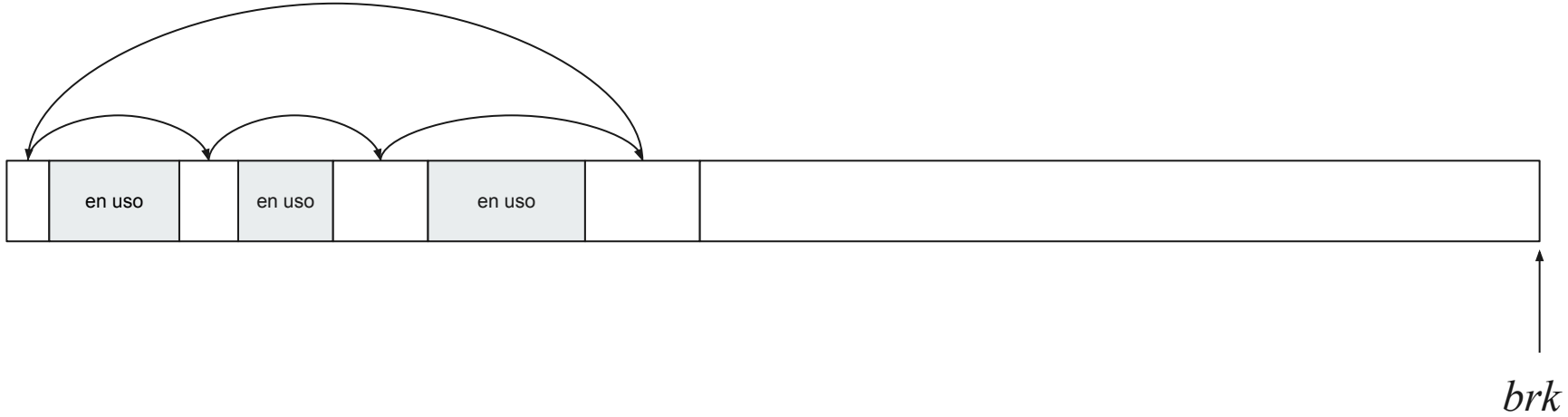
# Reservar memoria pidiendo mas al OS

```
ap = malloc(n)
```



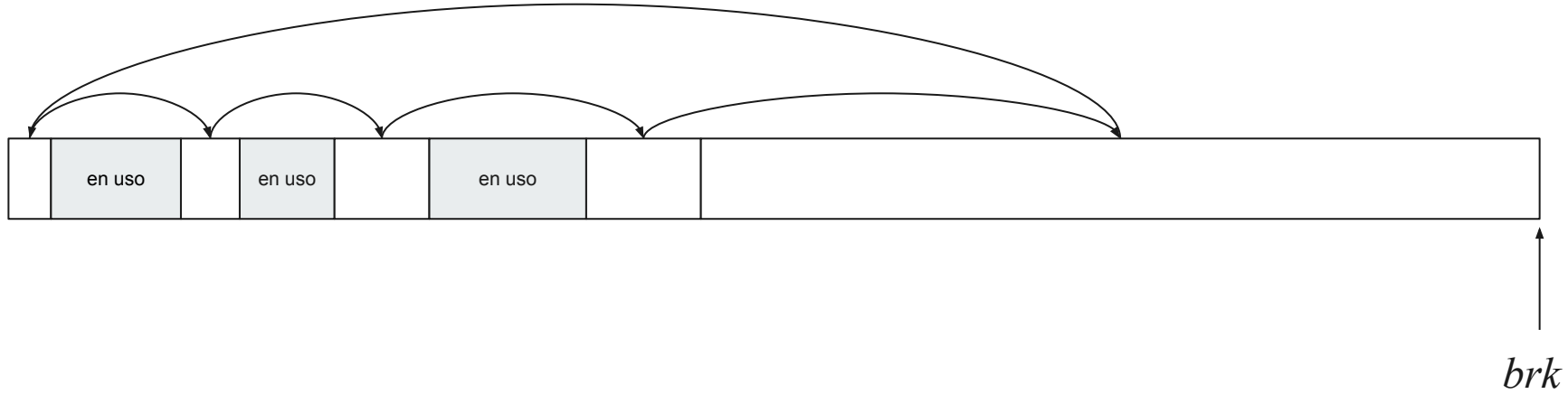
# Reservar memoria pidiendo mas al OS

```
ap = malloc(n)
```



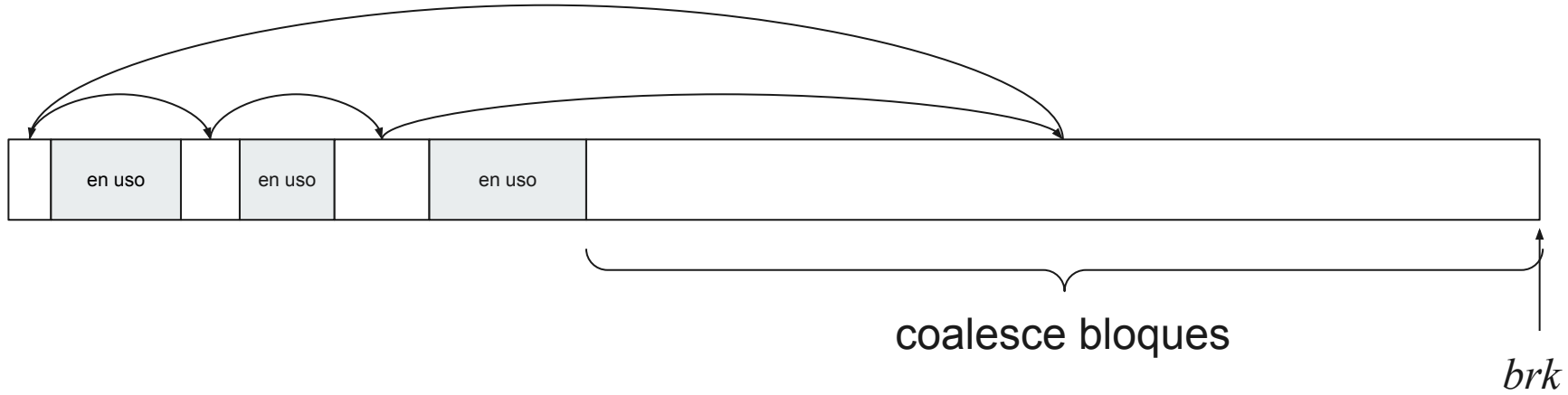
# Reservar memoria pidiendo mas al OS

```
ap = malloc(n)
```



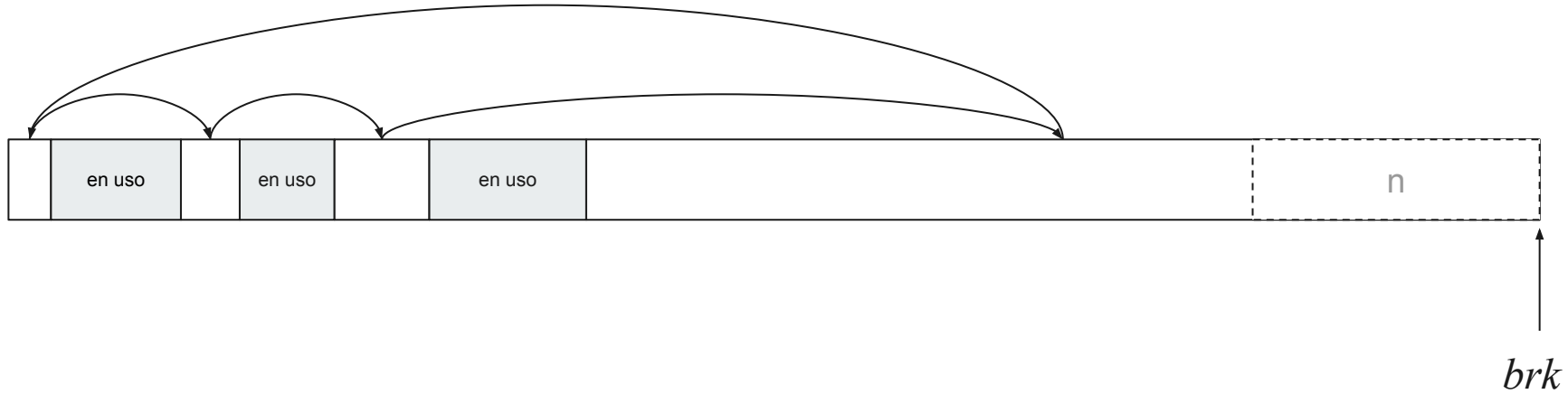
# Reservar memoria pidiendo mas al OS

```
ap = malloc(n)
```



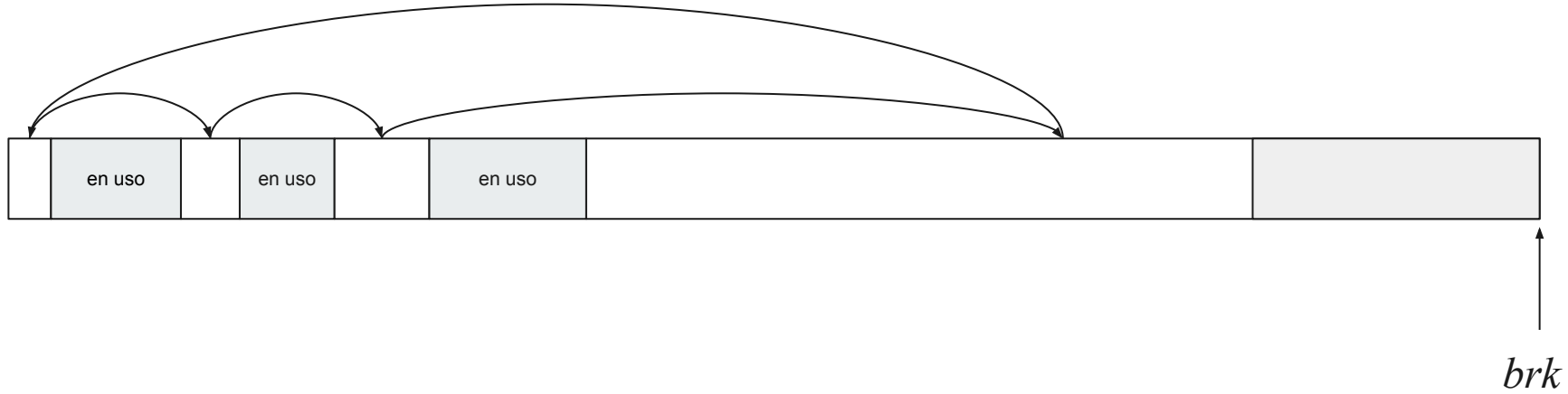
# Reservar memoria pidiendo mas al OS

```
ap = malloc(n)
```



# Reservar memoria pidiendo mas al OS

```
ap = malloc(n)
```





## mmap

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

<https://dashdash.io/2/mmap>



## Reservar un bloque con mmap

```
// mmap() returns a pointer to a chunk of free space

node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_ANONYMOUS|MAP_PRIVATE, -1,
0);

head->size = 4096 - sizeof(node_t);

head->next = NULL;
```





## MAP\_ANONYMOUS

The mapping is not backed by any file; its contents are initialized to zero. The *fd* argument is ignored [...] The *offset* argument should be zero.



# **Sistemas Operativos Concurrencia**

---

# Sincronización

## Conceptos clave:

- Race conditions
- Sección crítica

# Sincronización



La programación multihilo extiende el modelo secuencial de programación de un único hilo de ejecución. En este modelo se pueden encontrar dos escenarios posibles:

- Un programa está compuesto por un conjunto de threads independientes que operan sobre un conjunto de datos que están completamente separados entre sí y son independientes.
- Un programa está compuesto por un conjunto de threads que trabajan en forma cooperativa sobre un set de memoria y datos que son compartidos.

# Sincronización



En un programa que utiliza un modelo de programación de threads cooperativo, la forma de pensar secuencial no sirve:

1. La ejecución del programa depende de la forma en que los threads se intercalan en su ejecución, esto influye en los accesos a la memoria de recursos compartidos.
2. La ejecución de un programa es no determinística. Diferentes corridas pueden producir distintos resultados, por ejemplo debido a decisiones del scheduler. ¿Qué pasa con el debugging?
3. Los compiladores y el procesador físico pueden reordenar las instrucciones. Los compiladores modernos pueden reordenar las instrucciones para mejorar la performance del programa que se está ejecutando, este reordenamiento es generalmente invisible a los ojos de un solo thread

# Sincronización



Teniendo en cuenta lo anterior, la programación multithreading puede incorporar bugs que se caracterizan por ser:

- útiles
- no determinísticos
- no reproducibles

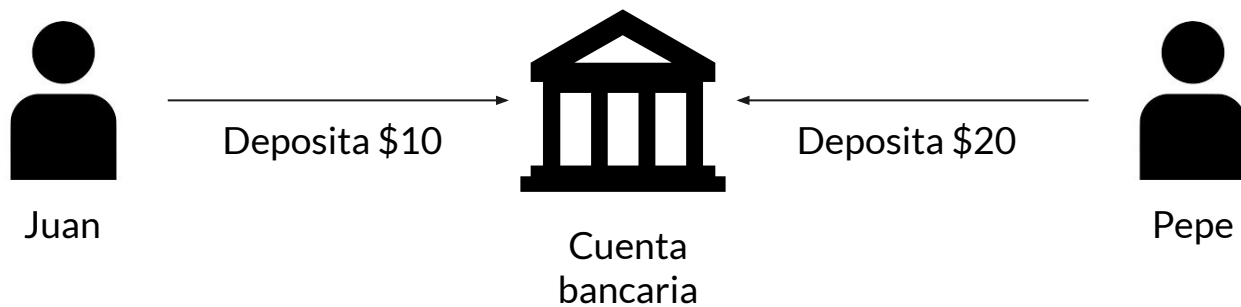
El approach a seguir en estos casos es: (1) estructurar el programa para que resulte fácil el razonamiento concurrente y (2) utilizar un conjunto de primitivas estándares para sincronizar el acceso a los recursos compartidos.

# Race Conditions



Una race condition se da cuando el resultado de un programa depende en como se intercalaron las operaciones de los threads que se ejecutan dentro de ese proceso. De hecho los threads juegan una carrera entre sus operaciones, y el resultado del programa depende de quién gane esa carrera.

## Ejemplo







## Ejemplo

<b>Thread 1: saldo = saldo + 10</b>	(saldo inicialmente vale 100)
<b>load      r1, saldo</b>	saldo=100 <b>r1=100</b>
<b>add      r1, r1, 10</b>	saldo=100 <b>r1=110</b>
<b>store    saldo, r1</b>	<b>saldo=110</b> r1=110



## Ejemplo

<b>Thread 2: saldo = saldo + 20</b>	(saldo inicialmente vale 100)
<code>load     r1, saldo</code>	saldo=100 <b>r1=100</b>
<code>add     r1, r1, 20</code>	saldo=100 <b>r1=120</b>
<code>store    saldo, r1</code>	<b>saldo=120</b> r1=120



## Ejemplo

load	r1, saldo	saldo=100 r1=100
add	r1, r1, 10	saldo=100 r1=110
store	saldo, r1	saldo=110 r1=110
load	r1, saldo	saldo=110 r1=110
add	r1, r1, 20	saldo=110 r1=130
store	saldo, r1	saldo= <b>130</b> r1=130



## Ejemplo

load     r1, saldo	saldo=100 r1=1
add     r1, r1, 20	saldo=100 r1=120
store    saldo, r1	saldo=120 r1=120
load     r1, saldo	saldo=120 r1=120
add     r1, r1, 10	saldo=120 r1= <b>130</b>
store    saldo, r1	<b>saldo=130</b> r1=130



## Ejemplo

load     r1, saldo	saldo=100 r1=100
add     r1, r1, 20	saldo=100 <b>r1=120</b>
load     r1, saldo	saldo=100 <b>r1=100</b>
store    saldo, r1	<b>saldo=100</b> r1=100
add     r1, r1, 10	saldo=100 <b>r1=110</b>
store    saldo, r1	<b>saldo=110</b> r1=110 <b>Se perdieron \$20!!!</b>

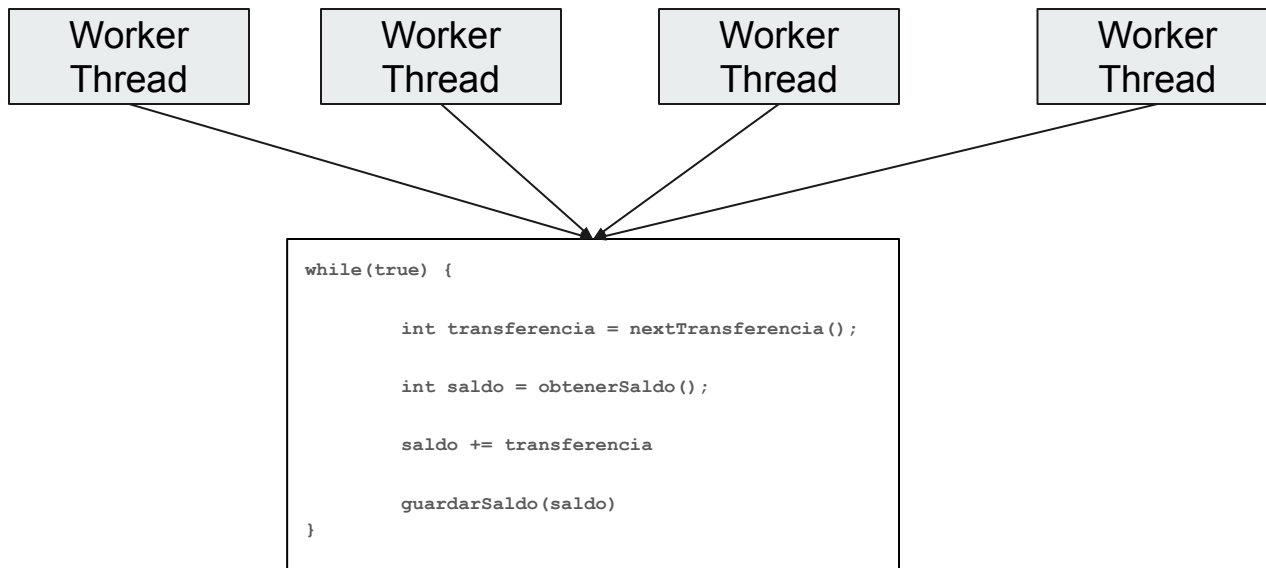


## Un típico proceso para procesar transacciones

```
while(true) {  
    int transferencia = nextTransferencia();  
    int saldo = obtenerSaldo();  
    saldo += transferencia  
    guardarSaldo(saldo)  
}
```



## Un típico proceso para procesar transacciones





## Sección Crítica

```
while(true) {  
  
    int transferencia = nextTransferencia();  
  
    int saldo = obtenerSaldo();  
  
    saldo += transferencia  
  
    guardarSaldo(saldo)  
  
}
```

Mientras que solo un thread a la vez  
entre a esta zona, esta todo bien!





## Sección Crítica

Es una abstracción, mediante la cual se define que en una sección de código solo puede haber un thread en ejecución a la vez.

Es una sección crítica define una sección de código que se ejecuta bajo “**exclusión mutua**”

---

# Locks

## Conceptos clave:

- Operaciones atómicas
- Mutex
- Spinlock y Sleeplock
- Deadlocks

# Lock

Un lock es una variable que permite la sincronización mediante la **exclusión mutua**, cuando un thread tiene el candado o lock ningún otro puede tenerlo.

La idea principal es que un proceso asocia un lock a determinados estados o partes de código y requiere que el thread posea el lock para entrar en ese estado. Con esto se logra que sólo un thread acceda a un recurso compartido a la vez.

Esto permite la exclusión mutua, todo lo que se ejecuta en la región de código en la cual un thread tiene un lock, garantiza la atomicidad de las operaciones.





# API de Locks

```
while(true) {  
    int transferencia = nextTransferencia();  
  
    obtener(lock)  
  
        int saldo = obtenerSaldo();  
  
        saldo += transferencia  
  
        guardarSaldo(saldo)  
  
    dejar(lock)  
}
```

## Lock contention

- Lock contention ocurre cuando múltiples threads o procesos intentan acceder a un recurso compartido simultáneamente, y al menos uno de ellos está bloqueado esperando un lock.



# Lock contention

```
while(true) {  
    int transferencia = nextTransferencia();
```

Entrar a la Seccion Critica



```
        int saldo = obtenerSaldo();  
  
        saldo += transferencia  
  
        guardarSaldo(saldo)
```

Salir de la seccion Critica

```
}
```



## API de Locks

```
pthread_mutex_t lock;  
int pthread_mutex_init(&lock, NULL);  
int pthread_mutex_lock (pthread_mutex_t * mutex);  
int pthread_mutex_unlock (pthread_mutex_t * mutex);  
int pthread_mutex_trylock(pthread_mutex_t * mutex);  
int pthread_mutex_timedlock(pthread_mutex_t * mutex, struct timespec  
*abb_timeout);
```



## Algunas Propiedades Formales

- **Exclusión mutua:** como mucho un solo Thread posee el lock a la vez.
- **Progress:** Si nadie posee el lock, y alguien lo quiere ... alguno debe poder obtenerlo.
- **Bounded waiting:** Si un thread quiere acceder al lock y existen varios threads en la misma situación, los demás tienen una cantidad finita (un límite) de posible accesos antes que T lo haga. Eventualmente el procesador le cede el lock a todos los que están esperando. Esta propiedad de los programas se llama **liveliness**





# Spinlock

Es un lock que se implementa usando un “busy wait”

Un “busy wait” es una técnica de programación en la que un proceso o hilo de ejecución espera activamente en un loop a que ocurra un evento en lugar de dormirse o bloquearse.

Esto significa que el proceso está ocupado en un bucle continuo verificando repetidamente si el evento ha ocurrido, sin realizar ninguna otra tarea útil en ese tiempo.



# Spinlock

Cuando se debe usar un spinlock (ie busy waiting)?

- Cuando el tiempo que se posee el lock es corto
- Cuando hay poca contention por los locks. Locking optimista, lo más probable es obtener el lock y no competir con otros threads.
- Cuando hay paralelismo real (múltiples procesadores). Sino no habrá ningún otro hilo realmente corriendo que pueda liberar el lock.



## Spinlock - xv6

```
struct spinlock {  
    uint locked;  
};
```



## Spinlock - xv6

```
void
acquire(struct spinlock *lk)
{
    [...]

    while(__sync_lock_test_and_set(&lk->locked, 1) != 0)
        ;

    [...]
}
```

# Operaciones Atómicas



Es una instrucción del procesador que se garantiza que se ejecuta completa: “todo o nada”.

- No puede haber una lectura parcial de valores, o race conditions
- No puede haber interrupciones entre la lectura y escritura a memoria, tampoco otros procesadores pueden modificar los valores de memoria entre la lectura y escritura de la instrucción.

La atomicidad es garantizada por diseño; la arquitectura del procesador define qué instrucciones son atómicas y la implementación del **hardware garantiza** que la instrucción es realmente atómica.

*Las más interesantes para nuestra discusión, son las instrucciones que hacen más de una cosa atómicamente.*



## Operaciones atómicas

```
type __sync_lock_test_and_set (type *ptr, type value, ...)
```

Este builtin, como lo describe Intel, no es una operación de test-and-set tradicional, sino más bien una operación de intercambio atómico. Escribe el valor en \*ptr y devuelve el contenido anterior de \*ptr.

Realiza todas las siguientes operaciones de manera atómica

1. Lee el valor actual de la variable.
2. Asigna un nuevo valor a la variable.
3. Devuelve el valor original de la variable.

Ref: <https://gcc.gnu.org/onlinedocs/gcc-4.5.3/gcc/Atomic-Builtins.html>



# Spinlock

```
void
release(struct spinlock *lk)
{
    [...]

    __sync_lock_release(&lk->locked);

    [...]
}
```



# Operaciones atómicas

```
void __sync_lock_release (type *ptr, ...)
```

Este builtin libera el lock adquirido por `__sync_lock_test_and_set`. Normalmente esto significa escribir la constante 0 en `*ptr`.

Ref: <https://gcc.gnu.org/onlinedocs/gcc-4.5.3/gcc/Atomic-Builtins.html>





## Sleep lock

Es un lock que se implementa cediendo el procesador.

- El proceso si no logra conseguir el lock se va a dormir.
- Cuando el lock es liberado los procesos esperando por el lock se despiertan

2 alternativas para despertarlos:

- El OS elige uno y lo despierta. Ese obtiene el lock.
- El OS despierta a todos y todos vuelven a competir para obtener el lock (xv6 hace esto)



# Spinlock

Cuando se debe usar un sleep lock?

- Cuando la espera será larga (eg. cuando se espera que el disco lea bloques) y no queremos bloquear el acceso al CPU para los otros procesos.
- En sistemas de un solo procesador, donde un spin-lock solo sería desalojado del procesador al acabarse su timeslice.



# Sleeplock

```
struct sleeplock {  
    uint locked;          // Is the lock held?  
    struct spinlock lk; // spinlock protecting this sleep lock  
};
```



# Sleeplock

```
void
acquiresleep(struct sleeplock *lk)
{
    acquire(&lk->lk);

    while (lk->locked) {
        // Atomically release lock and sleep on chan.
        // Reacquires lock when awakened.
        sleep(lk, &lk->lk);
    }
    lk->locked = 1;

    release(&lk->lk);
}
```



# Sleeplock

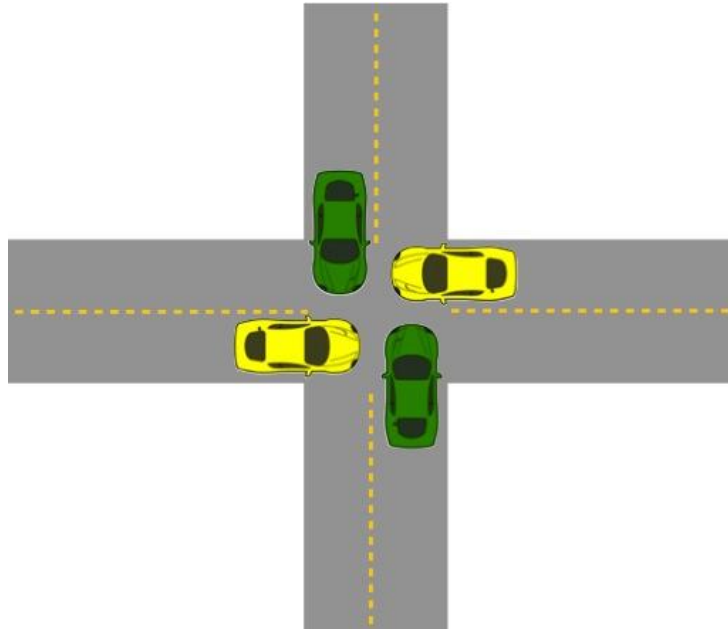
```
void
releasesleep(struct sleeplock *lk)
{
    acquire(&lk->lk);

    lk->locked = 0;
    // Wake up all processes sleeping on chan.
    wakeup(lk);

    release(&lk->lk);
}
```



# Deadlock






# Deadlock

Thread 1

```
lock (a)  
lock (b)  
.. hacer algo  
unlock (b)  
unlock (a)
```

Thread 2

```
lock (b)  
lock (a)  
.. hacer algo  
unlock (b)  
unlock (a)
```





# Deadlock

Thread 1

```
lock (a)
lock (b)
.. hacer algo
unlock (b)
unlock (a)
```

Thread 2

```
lock (a)
lock (b)
.. hacer algo
unlock (b)
unlock (a)
```