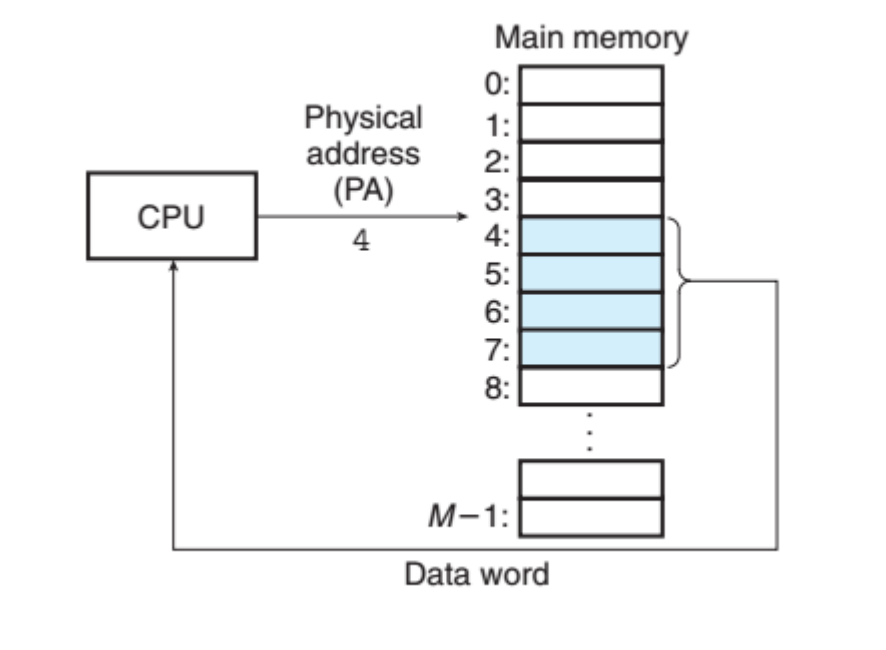
GESTIÓN DE MEMORIA Y DEBUGGING

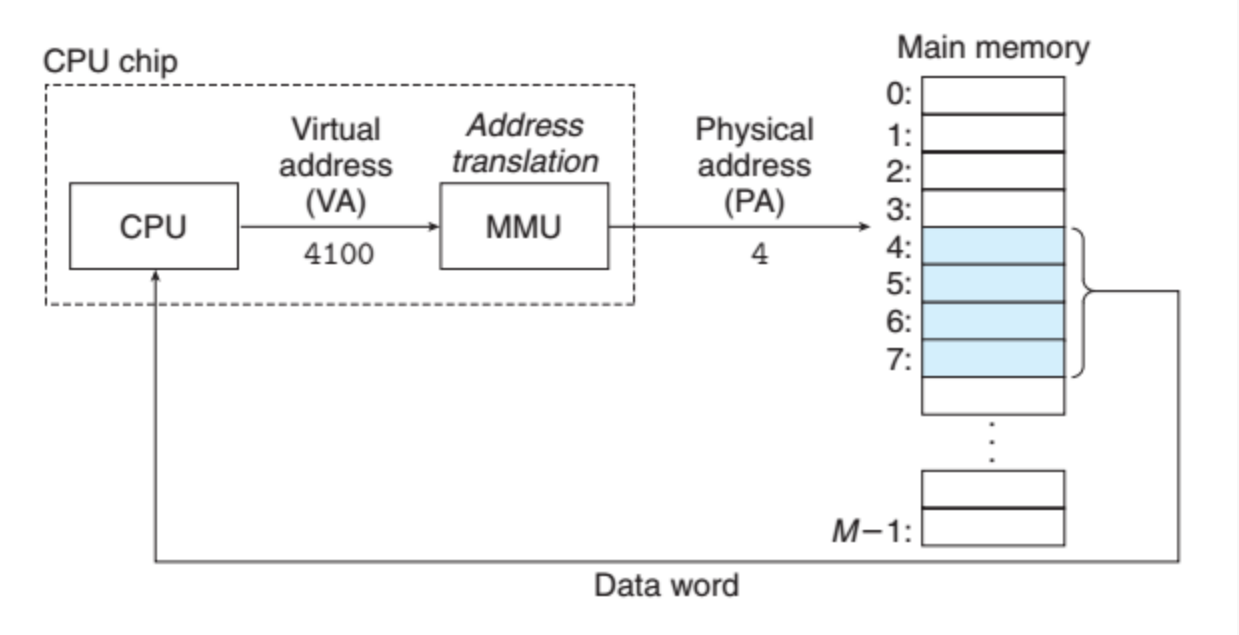
Gestión de Memoria y Debugging

Direccionamiento Físico



Gestión de Memoria y Debugging

Direccionamiento Virtual



Gestión de Memoria y Debugging

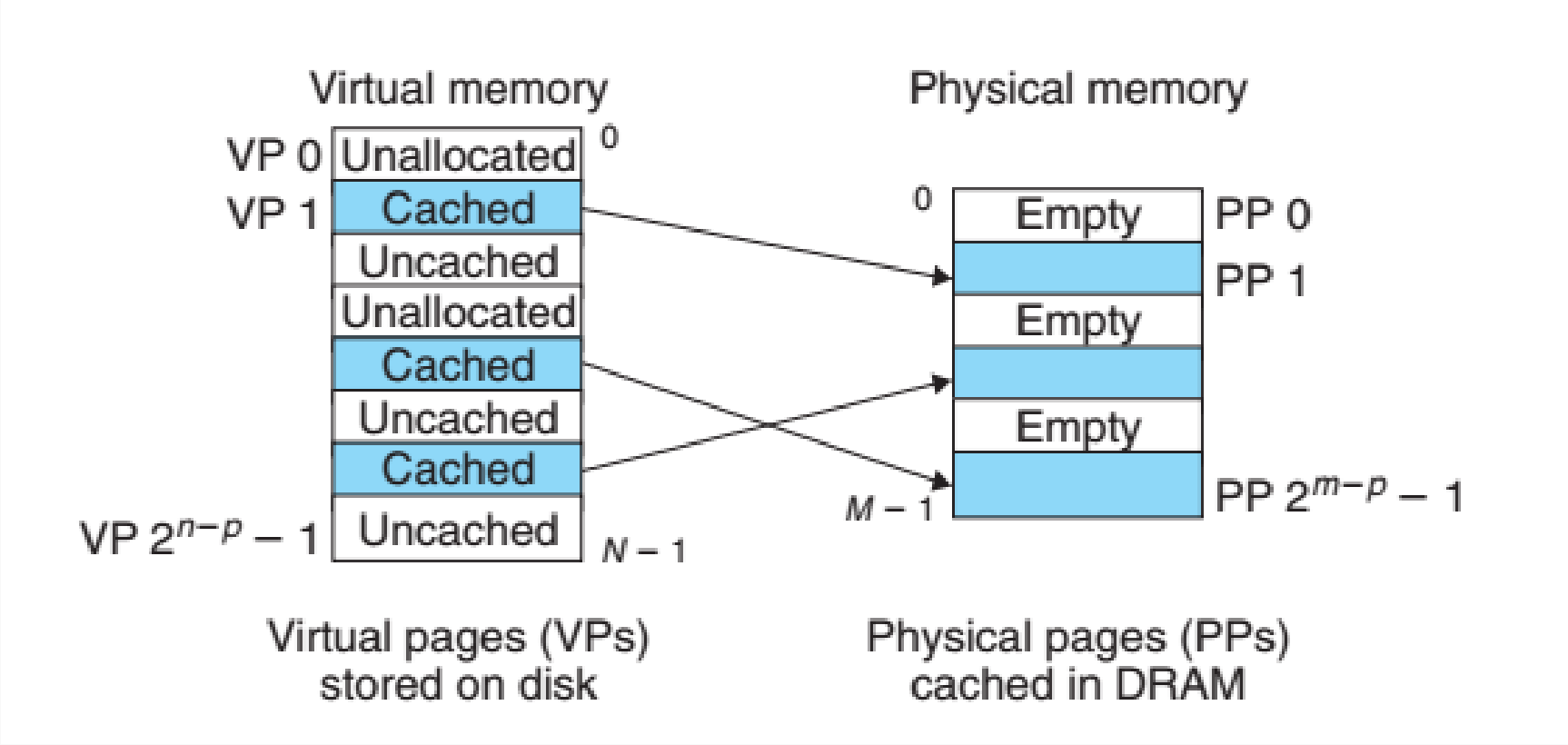
Espacio de direcciones

- Conjunto ordenado de direcciones enteras no negativas
- Si es consecutivo, lo llamamos **espacio de direcciones lineal**
- **En sistemas con memoria virtual**, CPU genera direcciones virtuales. A este le llamamos **ESPACIO DE DIRECCIONES VIRTUAL**

Gestión de Memoria y Debugging

- Un sistema también tiene el **ESPACIO DE DIRECCIONES FISICO**
- Estos permite diferenciar entre:
  - *Objetos (bytes)*
  - *Direcciones*
- Podemos hacer que un objeto tenga varias direcciones.

Gestión de Memoria y Debugging



Gestión de Memoria y Debugging

Memoria Virtual

- Arreglo de N celdas de 1 byte c/u almacenadas en disco.
- Cada byte tiene un VA (índice).
- Contenidos del arreglo se cachean en memoria física
- Datos en disco están particionados en unidades de tamaño fijo **PAGINAS VIRTUALES** (usualmente 4KB).

Gestión de Memoria y Debugging

- Memoria física está particionada en bloques del mismo tamaño

  PAGINAS FÍSICAS

- Páginas pueden estar
  - *No asignadas*
  - *Cacheadas*

Gestión de Memoria y Debugging

Cache de memoria DRAM

Disco es aprox. 100000 veces más lento que la memoria física.

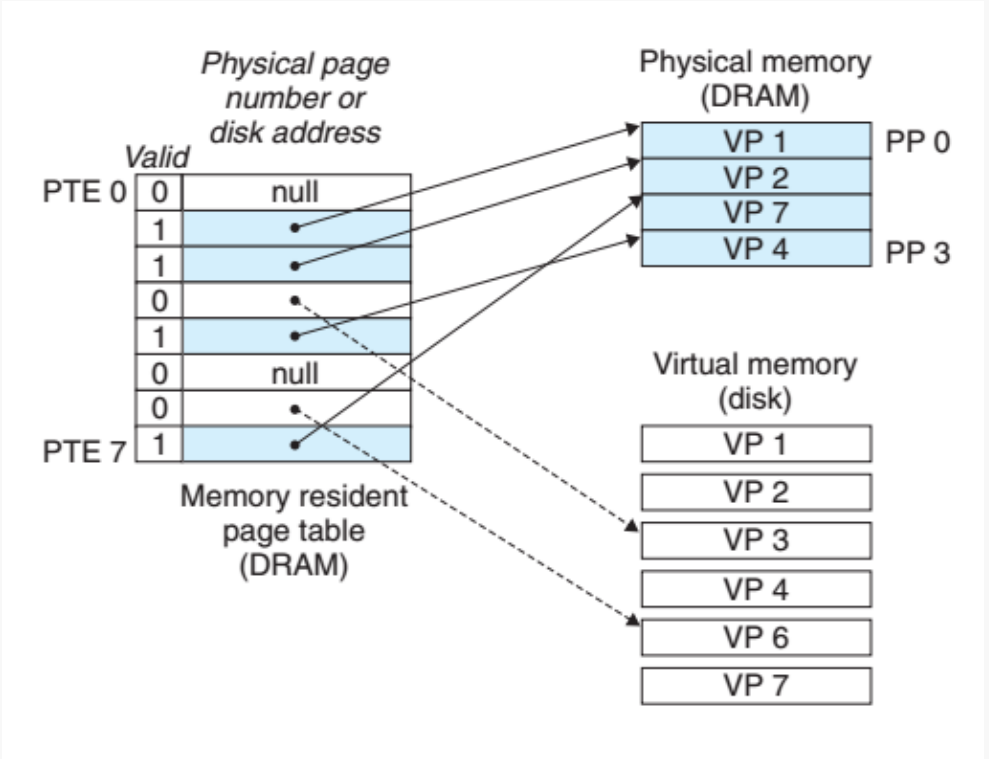
Nosotros queremos que la dirección que leemos, esté en un página física (RAM, DRAM cache). A esto le llamamos *hit*.

Si no está, tenemos que ir a disco. A esto le llamamos *miss*.

Gestión de Memoria y Debugging

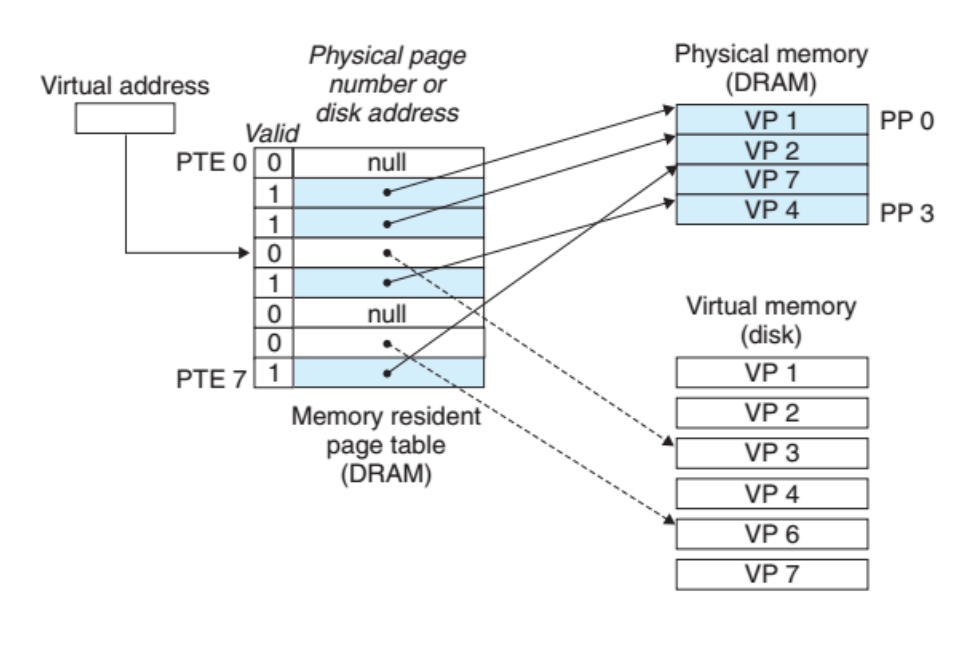
Tablas de página, page hit

Mapea paginas virtuales a páginas físicas. Hace uso de la MMU.

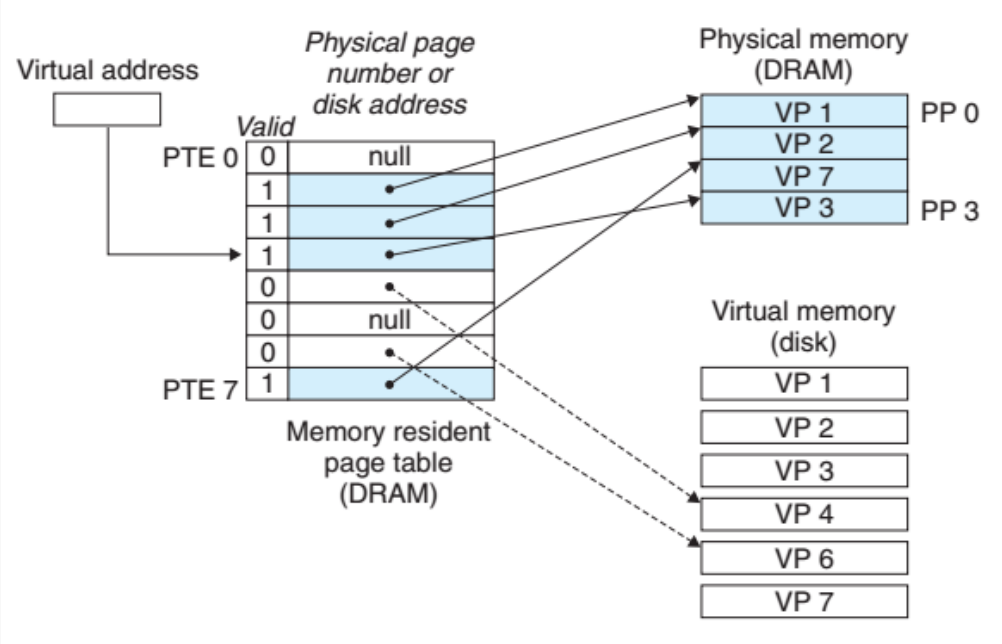


Gestión de Memoria y Debugging

Page fault (antes)

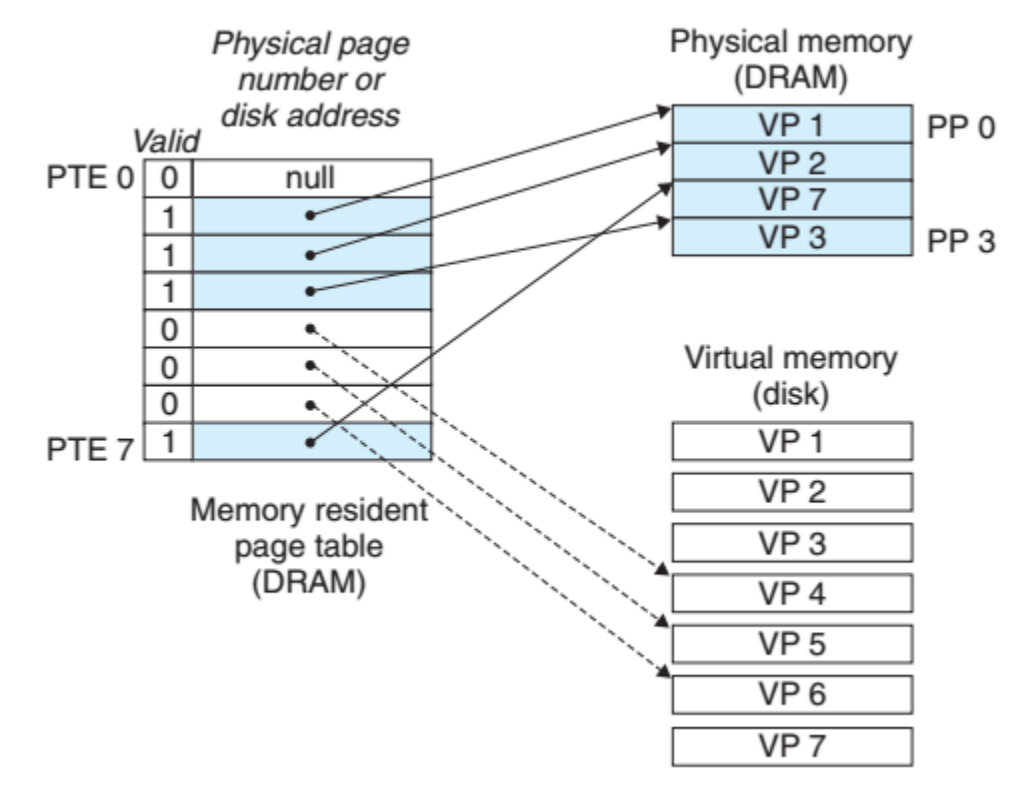


Page fault (después)



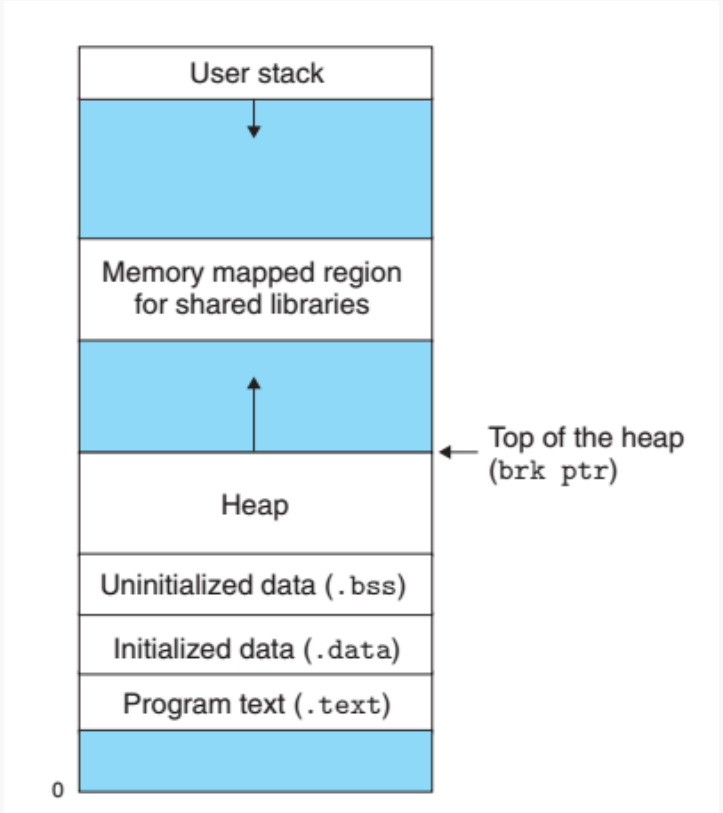
Gestión de Memoria y Debugging

Asignado una nueva página



Gestión de Memoria y Debugging

Asignación de memoria dinámica



Gestión de Memoria y Debugging

- El heap es usado por el *dynamic memory allocator* para mantener:
  - *Una colección de **bloques** de distintos tamaños.*
  - *Cada bloque es un pedazo contiguo de bytes que está:*
    - Asignado, o
    - Libre

Gestión de Memoria y Debugging

- Dos tipos de *allocators:*
  - ***Explicitos (C)***
  - ***Implicitos (Java, Python, etc).***

Gestión de Memoria y Debugging

- **Malloc y Free**

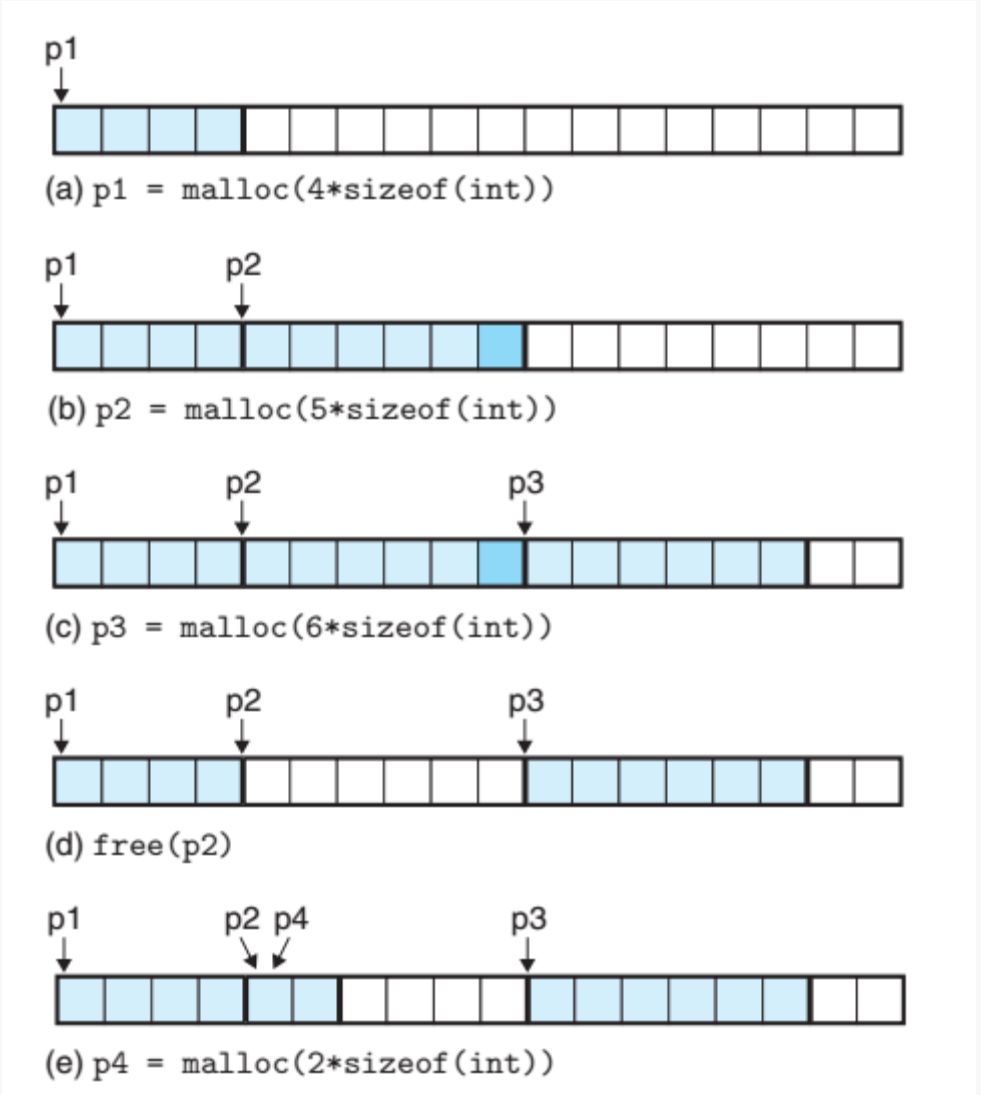
malloc retorna un **puntero** a un bloque de memoria de tamaño size, NULL si no puedo asignarlo:

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

Gestión de Memoria y Debugging

free libera un bloque de memoria asignado por malloc, calloc, realloc. Si el puntero no fue asignado por estás funciones, el comportamiento es indeterminado.

```
void free(void *ptr);
```


p1
(a) p1 = malloc(4*sizeof(int))
p1
p2
(b) p2 = malloc(5*sizeof(int))
p1
p2
p3
(c) p3 = malloc(6*sizeof(int))
p1
p2
p3
(d) free(p2)
p1
p2
p4
p3
(e) p4 = malloc(2*sizeof(int))

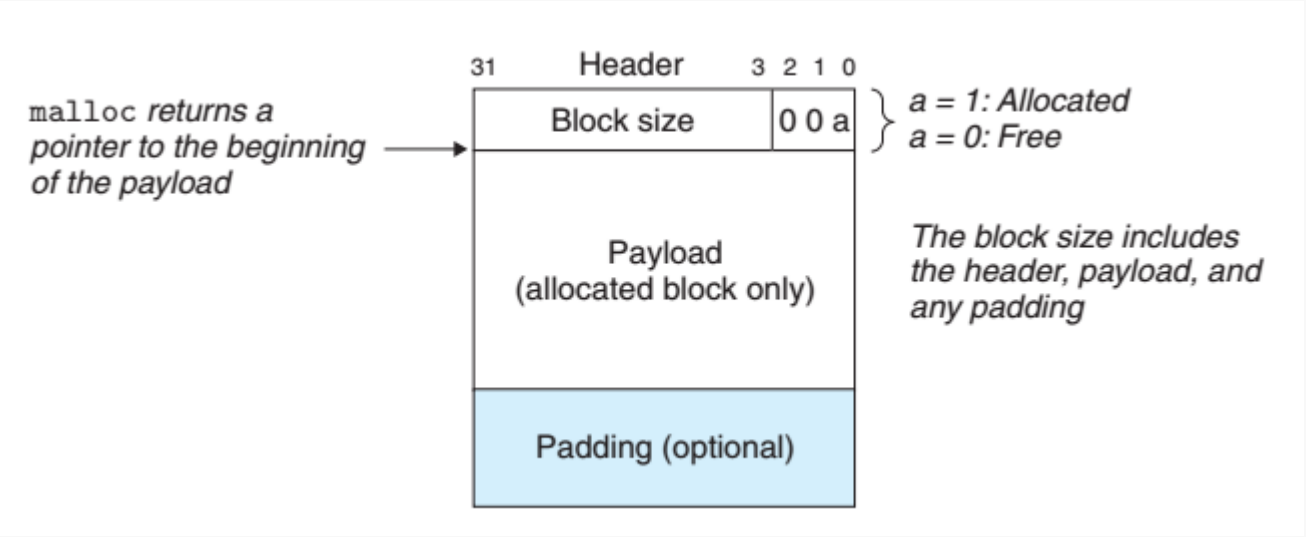
Gestión de Memoria y Debugging

Fragmentación interna y externa

- Interna: bloque asignado es más grande que la carga
- Externa: cuando hay espacio libre para cumplir solicitudes, pero no son contiguos

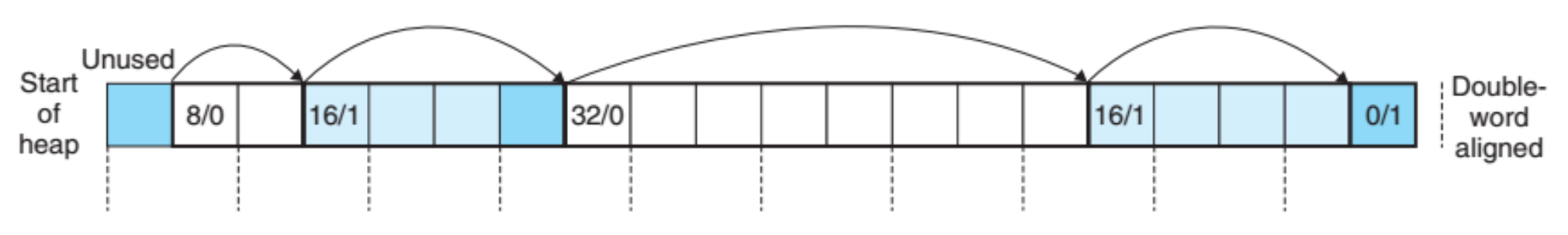
Gestión de Memoria y Debugging

Listas Libres Implícitas



Gestión de Memoria y Debugging

Listas Libres Implicitas



Gestión de Memoria y Debugging

Asignando bloques

Algoritmos:

- First fit
- Next fit
- Best fit

Gestión de Memoria y Debugging

Liberando memoria

Algoritmos:

- Liberar bloque entero (produce fragmentación interna)
- Dividir bloque en dos partes
  - *Una para ser asignada*
  - *La otra queda libre*

Gestión de Memoria y Debugging

Bugs de memoria típicos en C

- Punteros inválidos
- Leer memoria no incializada
- Overflow de stack (gets()!)
- Asumir que punteros y objetos a los que apuntan son del mismo tamaño

Gestión de Memoria y Debugging

Bugs de memoria típicos en C (continuación)

- Errores de aritmética de punteros
- Hacer referencia a variables inexistentes
- Hacer referencia a bloques liberados del heap
- Fugas de memoria (consumo incremental no controlado de memoria)

ENTRADA/SALIDA

Entrada/Salida

- Dos tipos
  - *Entrada/Salida sin buffer (unbuffered I/O)*
  - *Entrada/Salida estándar*
- Operaciones con ***unbuffered I/O*** se las manejar con cinco funciones
  - `open(2)`
  - `close(2)`
  - `read(2)`
  - `write(2)`
  - `lseek(2)`

Entrada/Salida

Descriptores de archivo

- Entero no negativo para referirse a archivo abierto
- Asignado cuando se llama a `open` o `creat`
- Se asigna el entero más pequeño disponible

- Tres descriptores importantes: `stdin`: descriptor 0, `stdout`: descriptor y `stderr`: descriptor 2

Entrada/Salida

- **Abriendo archivos**
  - *Dos llamadas:* `open` *y* `openat`
  - *Ambos retornan un descriptor de archivo, o* `-1` *en error.*

```
#include <fcntl.h>

int open( const char *path, int oflag, ... /* mode_t mode */);

int openat( int fd, const char *path, int oflag, ... /* mode_t mode */);
```

Entrada/Salida

Banderas

Requeridas

- O_RDONLY → abrir sólo lectura
- O_WRONLY → abrir sólo escritura
- O_RDWR → abrir para leer/escribir
- O_EXEC → abrir sólo como ejecutable
- O_SEARCH → abrir sólo para búsqueda (directorio)

Opcionales

- O_CREAT → crear archivo si no existe
- O_TRUNC → Truncar archivo
- O_SYNC → Hacer que write espere a que datos sean escritos a disco, en vez de solo encolarlos.
- O_APPEND → anexar al archivo
- Otras más…

Entrada/Salida

Creando archivos

- Usamos la función `creat`

```c
#include <fcntl.h>

int creat( const char *path, mode_t mode);
```

- Retorna un descriptor de archivo o -1 en error
- **Modos:** `S_IRWXU, S_IRUSR, S_IWUSR, S_IXUSR, S_IRWXG, S_IRGRP, S_IWGRP, S_IXGRP, S_IRWXO, S_IROTH, S_IWOTH, S_IXOTH`

Entrada/Salida

Cerrando archivos

- Usamos `close`

```
#include <fcntl.h>

int close(int fd);
```

- Cuando un proceso termina, todos sus descriptores de archivos abiertos son cerrados automáticamente.

Entrada/Salida

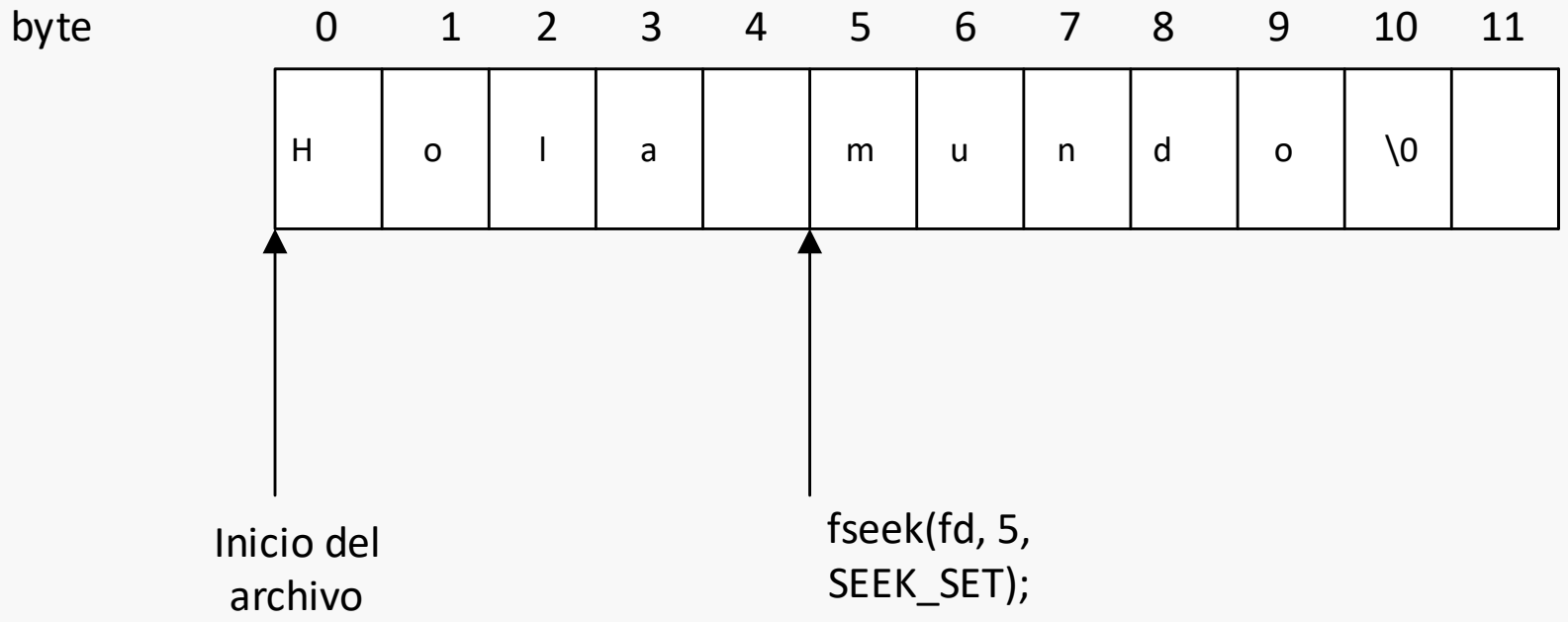
Avanzando dentro del archivo.

- La función lseek nos permite movernos dentro de esta secuencia de caracteres o bytes. SO mantiene un **cursor** con la posición actual dentro del archivo.

```
#include <unistd.h>

off_t lseek( int fd, off_t offset, int whence);
```

- `offset` dice **cuántos** bytes avanzamos. Puede ser negativo o positivo (usualmente positivo).
- `whence` dice **desde** donde empezamos a avanzar el cursor: SEEK_SET, SEEK_CUR, SEEK_END

Enrada/Salida



Entrada/Salida

Leyendo archivos

- Usamos la función `read`

```
#include <unistd.h>

ssize_t read( int fd, void *buf, size_t nbytes);
```

- Retorna el número de bytes leídos, o -1 en error
- Tenemos que proveerle un buffer donde almacenar la información leída.
- Especificamos cuántos bytes a leer.

Entrada/Salida

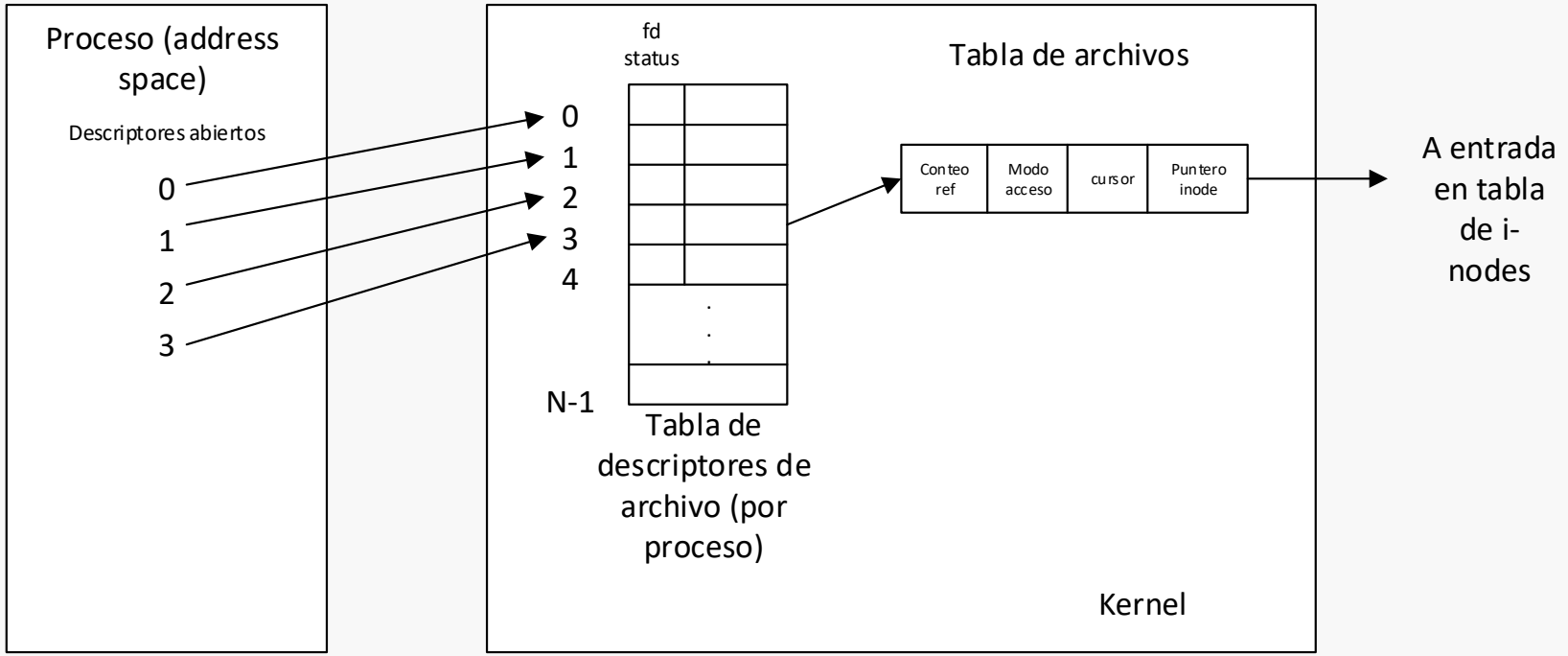
Escribiendo archivos

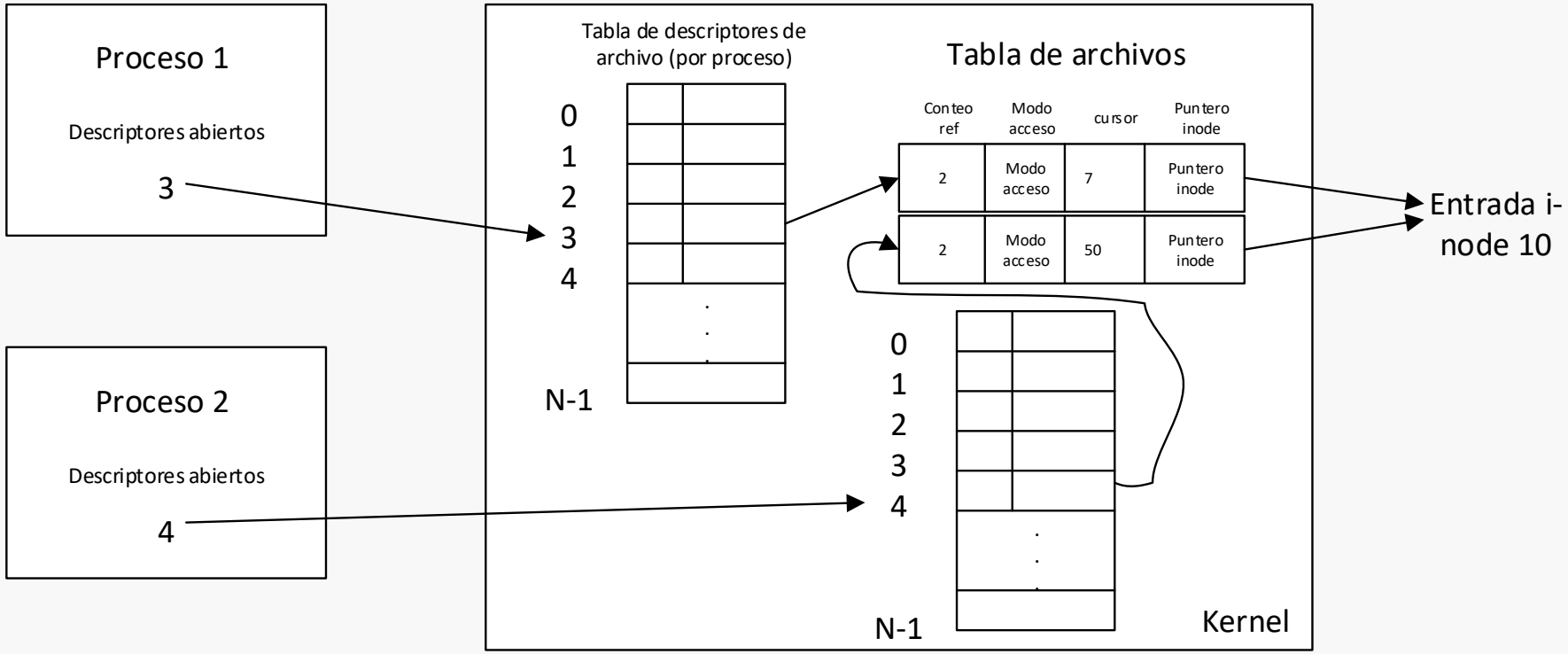
```
#include < unistd.h >

ssize_t write( int fd, const void*buf, size_t nbytes);
```

- Retorna el número de bytes escritos o -1 en error.
- Para archivo regular, escritura empieza desde la **posición actual del cursor**.
- Si archivo se abrió con `O_APPEND`, escritura empieza al final del archivo.

Entrada/Salida

Compartiendo Archivos




Proceso 1
Descriptores abiertos
3
Proceso 2
Descriptores abiertos
4
Tabla de descriptores de archivo (por proceso)
0
1
2
3
4
N-1
Tabla de archivos
Conteo ref
Modo acceso
cursor
Puntero inode
2
Modo acceso
7
Puntero inode
2
Modo acceso
50
Puntero inode
0
1
2
3
4
N-1
Kernel
Entrada i-node 10

Entrada/Salida

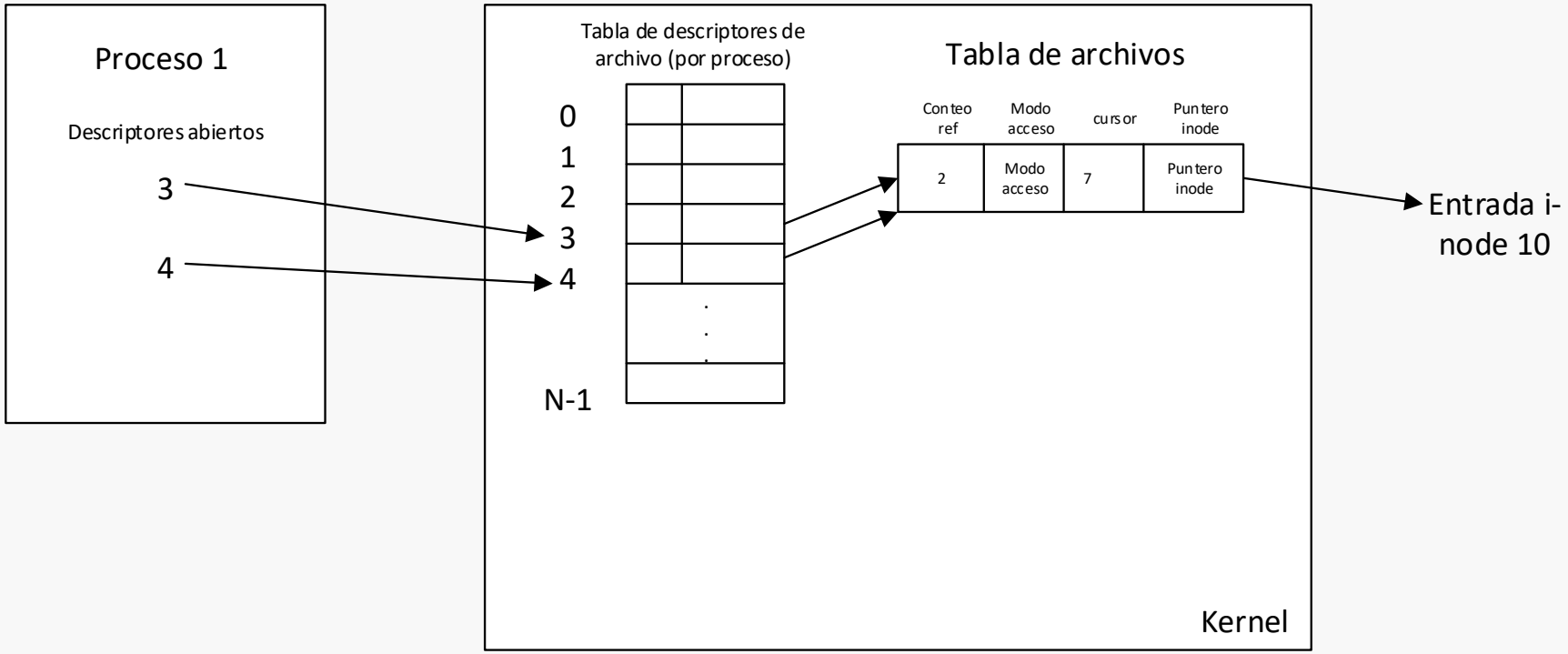
Funciones `dup` y `dup2`

Nos permiten duplicar descriptores de archivos

```
#include <unistd.h>

int dup( int fd);

int dup2( int fd, int fd2);
```

Ambas retornan el nuevo descriptor de archivo, que apunta al archivo referido por `fd`, o -1 si hubo error.

En `dup2`, especificamos el nuevo descriptor que queremos usar (`fd2`)


Proceso 1
Descriptores abiertos
3
4
Tabla de descriptores de archivo (por proceso)
0
1
2
3
4
N-1
Tabla de archivos
Conteo ref
Modo acceso
cursor
Puntero inode
2
Modo acceso
7
Puntero inode
Entrada i-node 10
Kernel