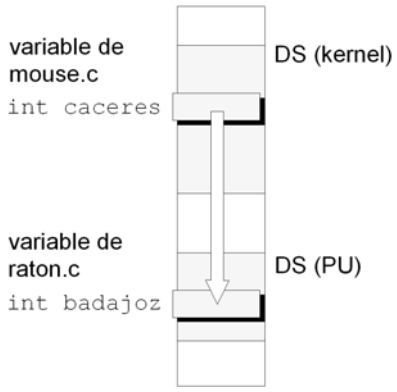
Práctica 4: Transferencia de datos entre el núcleo y los procesos de usuario.

El objeto de esta práctica es dotar de un servicio de lectura a un manejador de dispositivo y utilizarlo desde un programa de usuario mediante la función UNIX `read`. El guión de práctica es el que sigue:

1. Introducción.



La figura muestra una copia entre dos espacios de direccionamiento distintos. El origen es la variable entera `caceres` del fichero fuente del nuevo manejador de dispositivo, `mouse.c`. Como los manejadores de dispositivo en Minix 2.0.2 forman parte del núcleo, `caceres` se ensambla en el segmento de datos de este. El destino de la copia es la variable entera `badajoz`, del fichero fuente `raton.c`, el programa de usuario que verifica que el nuevo manejador funciona correctamente.

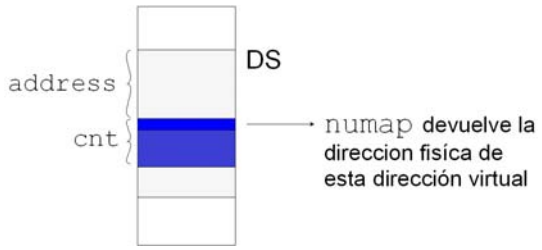
El servicio de lectura que vamos a proporcionar en esta práctica es muy simple. No se leen datos de ningún dispositivo. Simplemente permite al programa de usuario inspeccionar el valor de la variable del kernel `caceres` mediante la llamada `read`, por ejemplo:

```
read(df, &badajoz, 4);
```

Tras esta llamada, la variable `badajoz` se debe haber actualizado con el valor de `caceres`:

2. Direcciones virtuales y direcciones físicas. Utilidades de copia.

Para el compilador de C un identificador de variable como `caceres` no es más que la dirección de memoria de dicho objeto. Como el compilador de C genera código reubicable, esta dirección no es absoluta, sino relativa al comienzo del segmento de datos del programa, según muestra la figura. A estas direcciones las denominaremos *virtuales*. Para que el nuevo manejador pueda hacer la transferencia de la variable `caceres` al programa de usuario que la solicita, sin embargo, necesita las direcciones *físicas* tanto de la fuente como del destino.



Observemos la figura. El núcleo determina la dirección física de una dirección virtual `address` de un proceso dado `proc_nr` mediante la función de utilidad `numap`. Su prototipo se declara en el fichero `/usr /src/kernel/proto.h`:

```
phys_bytes numap(int       proc_nr,
                 vir_bytes address,
                 vir_bytes cnt);
```

Devuelve 0 en caso de error y la dirección absoluta en otro caso. El argumento `proc_nr` es el número del proceso en cuestión. En la clase de teoría hemos visto que varía entre el número de tarea más negativo, `TTY`, y el proceso de usuario más alto. El gestor de memoria toma el número de proceso 0 y el sistema de ficheros el 1. El argumento `cnt` no es realmente necesario. Le sirve al núcleo en algunos casos para determinar que el bloque del que requerimos su dirección física reside completamente en el espacio de usuario parámetro.

El núcleo determina la dirección física de cualquiera de sus propias direcciones virtuales mediante la macro `vir2phys`:

```
phys_bytes vir2phys(vir_bytes address);
```

Finalmente, presentamos la utilidad de copia entre espacios de direccionamiento `phys_copy`. Su prototipo se declara en el fichero `/usr /src/kernel/proto.h`:

```
phys_bytes phys_copy(phys_bytes src, phys_bytes dst, phys_bytes cnt);
```

3. Mensajes de petición y réplica.

El mensaje de petición que el sistema de ficheros envía a la tarea es de tipo `DEV_READ`. Tiene los siguientes campos:

Campo	Significado
m_type	Tipo del mensaje: `DEV_READ`
PROC_NR	Número del proceso de usuario que invocó `read`
ADDRESS	Dirección virtual destinataria. Es el segundo parámetro de `read`
COUNT	Número de octetos que se desean leer. Es el tercer parámetro de `read`

El mensaje de réplica que la tarea envía al sistema de ficheros tiene los siguientes campos:

Campo	Significado
m_type	Tipo del mensaje. La tarea lo rellena con `TASK_REPLY`
REP_PROC	La tarea lo rellena con el número del proceso de usuario que invocó `read`
REP_STATUS	Código de retorno. Informa al sistema de ficheros de si la petición se realizó correctamente. Si es así, se rellena con `OK`

4. Extendiendo el manejador de dispositivo.

Modificación 1.
Se trata de la tercera versión de `/usr/src/kernel/mouse.c`. Esta vez el trabajo consiste en implementar el caso `DEV_READ`. Se trata de copiar el contenido de la variable local `caceres` a la variable del proceso de usuario que invocó `read`, según se ha indicado en el punto 1.

```
#include "kernel.h"
#include <minix/com.h>
#include <minix/callnr.h>
void mouse_task()
{
  message mess;
  int     status;
  int     caceres = 5;
  while(1) {
    receive(ANY, &mess);
    switch(mess.m_type) {
      case DEV_OPEN:
        printf("...");
        break;
      case DEV_CLOSE:
        printf("...");
        break;
      case DEV_READ:
        ...
        break;
    }
    mess.m_type      = TASK_REPLY;
    mess.REP_PROC_NR = mess.PROC_NR;
    mess.REP_STATUS  = OK;
    if(OK != (status = send(mess.m_source, &mess)))
      panic("Error en mouse\n", status);
  }
};
```

5. Escribiendo el programa de usuario.

Modificación 2.

Finalmente, ampliaremos el programa de usuario `raton.c`, que prueba el nuevo servicio de lectura.

```
! cd /usr/home
! mined raton.c
```

```c
#include <sys/types.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
main()
{
  int fd;
  int badajoz = 0;
  if(0 > (fd = open("/dev/mouse", 0)) {
    perror("Raton: error en open");
    exit(1);
  }
  printf("Raton abierto\n");
  if(0 > read(fd, &badajoz, 4)) {
    perror("Raton: error en read");
    exit(1);
  }
  if(0 > close(fd)) {
    perror("Raton: error en close");
    exit(1);
  }
  if(badajoz != 5)
    printf("Algo fue mal: Badajoz = %d\n", badajoz);
  else {
    printf("Correcto: Badajoz = %d\n", badajoz);
    printf("Practica superada con exito\n");
  }
}
```

```
! cc raton.c -o raton
! raton
```