PRÁCTICA 4

paginación
en Minix 3.1.2a

Pedro Pablo López Rodríguez

ÍNDICE

1 OBJETIVOS

Esta práctica está orientada a que el alumno consolide los conocimientos teóricos que se estudian en clase en el tema de Memoria (trasparencias 36 en adelante) relativos a la paginación, aplicándolos sobre un sistema real como Minix que se ejecuta sobre la arquitectura del 80386. En dicha arquitectura está disponible el mecanismo de paginación con tablas de páginas a dos niveles (Memoria trasparencia 48) por lo que el alumno aprenderá a manipular este tipo de tablas de páginas, lo que deberá hacerse dentro del núcleo. Por tanto la inicialización de la tabla de páginas a dos niveles y la activación de la paginación implicarán la modificación del fichero ya conocido por el alumno desde la práctica1: **/usr/src/kernel/main.c**.

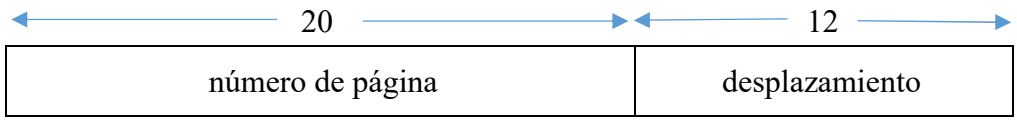
2 CARACTERÍSTICAS DE LA PAGINACIÓN A UTILIZAR

El microprocesador 80386 es un procesador de 32 bits, por lo que es perfectamente capaz de generar direcciones virtuales de 32 bits que pueden abarcar un espacio de direccionamiento virtual de 2^{32} Bytes = $2^2 \times 2^{30}$ Bytes = **4 GBytes**.

La máquina qemu emula un ordenador con un microprocesador 80386 y un tamaño de memoria física que podemos configurar. Vamos a suponer que dicho tamaño está configurado en (minix3.conf) a un tamaño de **32 MBytes** = $2^5 \times 2^{20}$ Bytes = 2^{25} Bytes.

El tamaño de las páginas en el 80386 es de **4 KB** = $2^2 \times 2^{10}$ Bytes = 2^{12} Bytes.

Por tanto el formato de una dirección virtual (de 32 bits) es: los 12 bits de menor peso corresponden al desplazamiento (offset) de la dirección dentro de la página, y los 20 bits restantes corresponden al número de página que contiene dicha dirección virtual:



Por ejemplo si tenemos la dirección virtual 0x1234abcd el número de la página que ocupa sería 0x1234a, mientras que 0xbcd sería el desplazamiento de esa dirección dentro de la página que ocupa (ya que cada dígito hexadecimal se corresponde con una combinación de 4 bits).

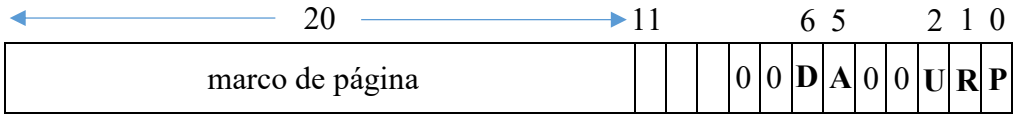
Como ya sabemos de la teoría, el espacio de direccionamiento virtual está dividido en páginas y el espacio de direccionamiento físico está dividido en marcos de página del mismo tamaño que las páginas (4 KB). En consecuencia el número de páginas y de marcos sería:

número de páginas = 4 GB / 4 KB = 2^{20} páginas

número de marcos = 32 MB / 4 KB = 8 K marcos

En el 80386 la paginación es a dos niveles, lo que quiere decir que la tabla de páginas está fraccionada en múltiples **subtablas de páginas** (tablas de páginas de segundo nivel) que ocupan un marco (de 4KB) cada una de ellas. Además hay un **directorio de páginas** (tabla de páginas de primer nivel) que nos dice qué marco ocupa cada subtabla de páginas.

Las entradas de la tabla de páginas (**descriptores de página**) en el 80386 tienen un tamaño de 32 bits con el siguiente formato:



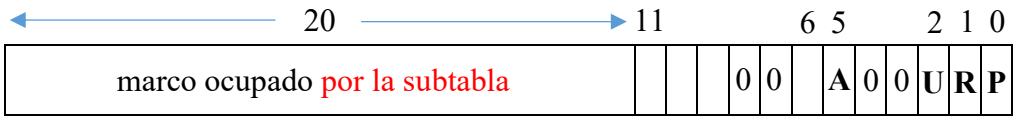
P: bit de página presente (1) o ausente (0)
R/W: read/write (0: read only, 1: se puede leer y escribir) bit de protección
U/S: usuario/supervisor (0: supervisor only, 1: usuario)
A: página accedida (bit de página referenciada)
D: página sucia (bit de página modificada)

Por ejemplo si el descriptor de la página 0xabcde es **0x12345000** eso nos diría que la página 0xabcde no está cargada (presente) en ningún marco. Si por el contrario el descriptor fuera 0x12345001 eso significaría que la página 0xabcde está presente en el marco 0x12345 y por tanto las direcciones virtuales desde la 0xabcde000 a la 0xabcdefff se corresponden con las direcciones físicas 0x12345000 a 0x12345fff. Además la página puede accederse sólo para lectura, y puede ser accedida sólo en modo supervisor (núcleo). Finalmente la página no ha sido modificada desde su carga en el marco 0x12345, pero además no se ha accedido a ella nunca ni siquiera para leerla. [**Ejercicio:** ¿Qué indicaría 0x12345067 como descriptor de página?]

Como cada página tiene asociado su descriptor de página, la tabla de páginas debe tener 2^{20} entradas. Como cada entrada ocupa 32 bits (4 Bytes) cada marco de página puede contener exactamente 4 KB / 4 B = 1024 entradas. Es decir cada subtabla de páginas (o tabla de segundo nivel) contiene 1024 entradas correspondientes a 1024 páginas.

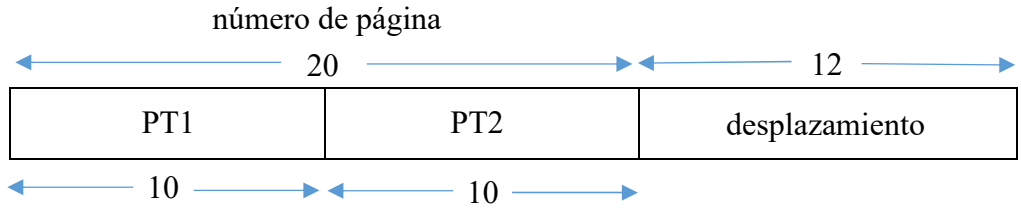
En consecuencia las 2^{20} entradas totales de la tabla de páginas requieren $2^{20}/1024 = 1024$ subtablas de páginas.

El directorio de páginas (la tabla de páginas de primer nivel) lleva la cuenta de las 1024 subtablas mediante descriptores con un formato análogo al descriptor de página:



La salvedad es que el descriptor es relativo a la correspondiente subtabla, la cual puede estar presente o no, No obstante para ver la descripción oficial del formato de los descriptores de página y de las páginas de segundo nivel lo mejor es consultar la página 102 del manual de Intel: 64-ia-32-architectures-software-developer-system-programming-manual.pdf disponible en "X:\Material\Bibliografia\Intel® 64 and IA-32 Architectures Software Developer Manuals".

El directorio de páginas es una tabla de 1024 entradas que debe estar cargada completamente en la memoria física y ocupa un único marco ya que (1024 × 4 B) / 4 KB = 1. Cada entrada del directorio de páginas indica si la subtabla correspondiente está presente o no, y en caso de estarlo nos dice en qué marco se encuentra cargada. Por tanto para traducir una dirección virtual con la tabla de páginas a dos niveles hay que dividir el número de página (20 bits) en los 10 bits de mayor peso para indexar el directorio de páginas, y los diez bits restantes para indexar la subtabla de páginas indicada en el descriptor de la subtabla obtenido, de manera que obtengamos de ella el descriptor de la página, y por tanto el marco que ocupa esa página. Añadiendo el campo de desplazamiento de la dirección virtual obtendríamos la dirección física correspondiente a la dirección virtual de partida.



La siguiente figura tomada de la página 98 del manual de Intel resume el proceso de traducción a la vez que nos indica que el registro de control **CR3** del 80386 (cuando la paginación está activada) contiene la dirección física del directorio de páginas.

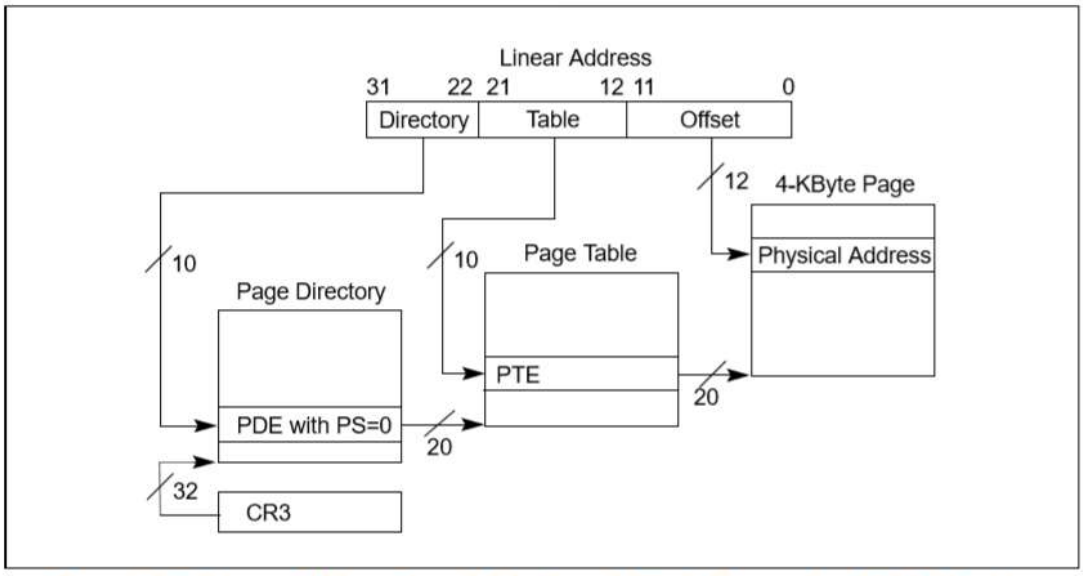


Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

En resumen, para el 80386:

- El directorio de páginas contiene 1024 entradas y ocupa un marco (4 KB)
- CR3 debe apuntar a la dirección física de comienzo del directorio de páginas.
- Cada entrada del directorio de páginas debe indicar el marco en el que está cargada la correspondiente subtabla (si está presente, es decir P = 1)
- Cada subtabla contiene 1024 entradas y ocupa un marco (4 KB)
- Cada entrada de la subtabla debe indicar el marco en el que está cargada la correspondiente página (si está presente, es decir P = 1)

En cuanto a la memoria principal ocupada por la tabla de páginas a dos niveles completa, sería 4 KB del directorio de páginas y 1024 × 4 KB = 4 MB ocupados por las 1024 subtablas.

3 INICIALIZACIÓN DE LA TABLA DE PÁGINAS

Con el fin de que el alumno aprenda a trabajar con una tabla de páginas a dos niveles en Minix vamos a proponer su inicialización de la manera más sencilla posible que es mapeando toda dirección virtual en la dirección física coincidente en valor con la dirección virtual de partida, es decir df = dv. De esta manera cuando se active la paginación en **/usr/src/main.c** todo seguirá funcionando sin problemas, tanto el núcleo como los procesos de usuario que compartirán todos el mismo espacio de direccionamiento virtual.

No obstante en nuestro caso (máquina virtual qemu para Minix) hemos dicho que la memoria física es de 32 MB, por lo que sabemos que normalmente Minix no va a utilizar direcciones mas allá de los 32 MB. Por tanto la función de traducción de direcciones dv → df que realmente queremos establecer es:

df = **si** (dv < 32 M) [es decir FT(dv) = dv]
dv
en otro caso
excepción (de falta de subtabla o de página) !!!!!

Para poder traducir los 32 MB primeros necesitamos 32 MB / 4 KB = 8 K descriptores de página. Esos 8 K descriptores caben en (8 K × 4 B) / 4 KB = 8 subtablas (de la 0 a la 7). Por tanto en el directorio de páginas las 8 primeras entradas apuntarán a esas 8 subtablas y las entradas de esas subtablas apuntarán al marco con el mismo valor que la página correspondiente a la entrada de la subtabla.

Para el resto de direcciones por encima de los 32 MB no es necesaria ninguna subtabla, sino que bastará con indicar en las correspondientes entradas del directorio de páginas (de la 8 a la 1023) que las subtablas no están presentes (P = 0).

Como lo anterior parece un trabalenguas de mucho cuidado, vamos a expresarlo de manera mas concisa en C, para lo cual establecemos las siguientes declaraciones:

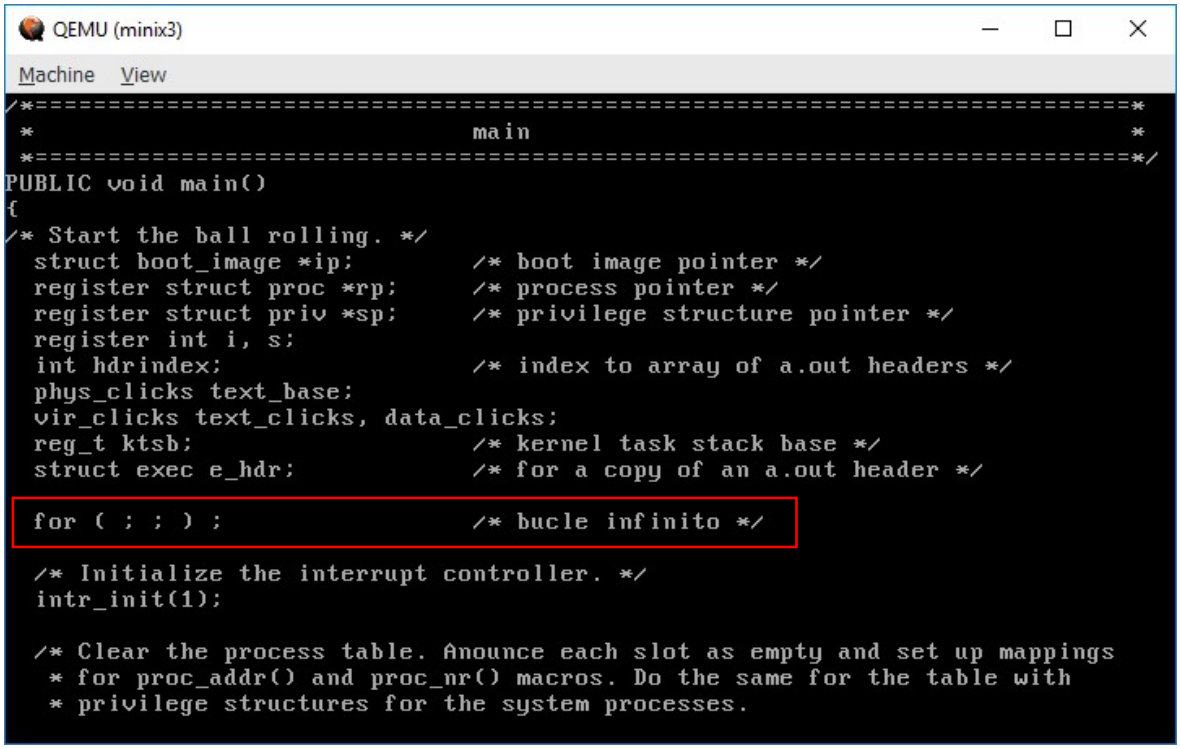
```
typedef unsigned descriptor_t ;                    /* 32 bits en Minix 3 */

descriptor_t DP [ 1024 ] ;                       /* directorio de paginas */

descriptor_t TP [ 8 * 1024 ] ;                /* 8 (sub)tablas de paginas */
```

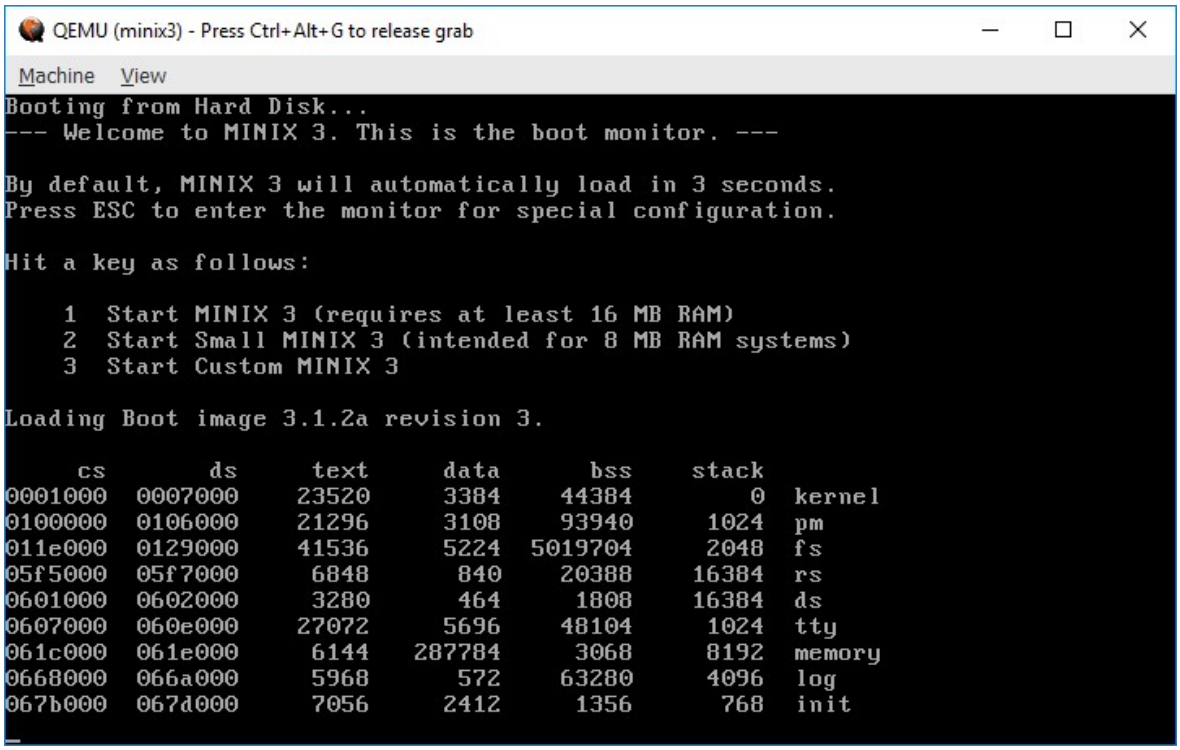
- Programe la inicialización de esas dos tablas (arrays) para que la función de traducción correspondiente a esa tabla de páginas a dos niveles sea la especificada anteriormente.

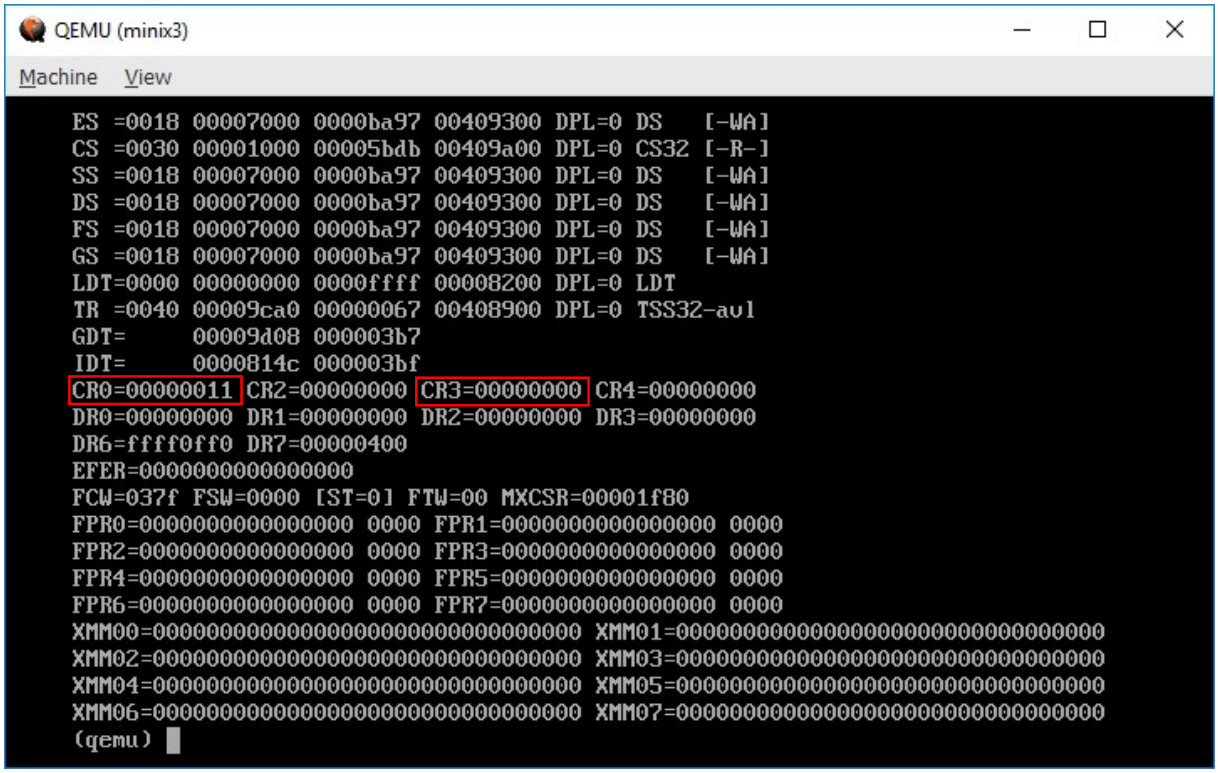
Incluya el código en la función main de **/usr/src/kernel/main.c** y compile el fichero con **make** para comprobar que al menos no hay errores sintácticos, los cuales deben corregirse completamente.

4 ACTIVACIÓN DE LA PAGINACIÓN

Una vez que el alumno tiene ya una idea de cómo se inicializa la tabla de páginas a dos niveles es el momento de explicar cómo se activa la paginación. La respuesta es sencilla dado que cuando se ejecuta la función main de **/usr/src/kernel/main.c** el procesador está ya en modo protegido como puede comprobarse inspeccionando los registros con el comando del monitor de qemu info registers (y tras haber puesto un bucle infinito en la función main):







Vemos que el bit 0 de CR0 está a 1 lo que indica que estamos en modo protegido. Por el contrario el bit 31 de CR0 está a 0 lo que indica que no está activada la paginación. Por otro lado el registro CR3 no contiene la dirección de comienzo del directorio de páginas que queremos utilizar. Puede consultarse la función de estos registros de control en el manual de Intel en sus páginas 63 y 64, de donde hemos extraido la siguiente figura:

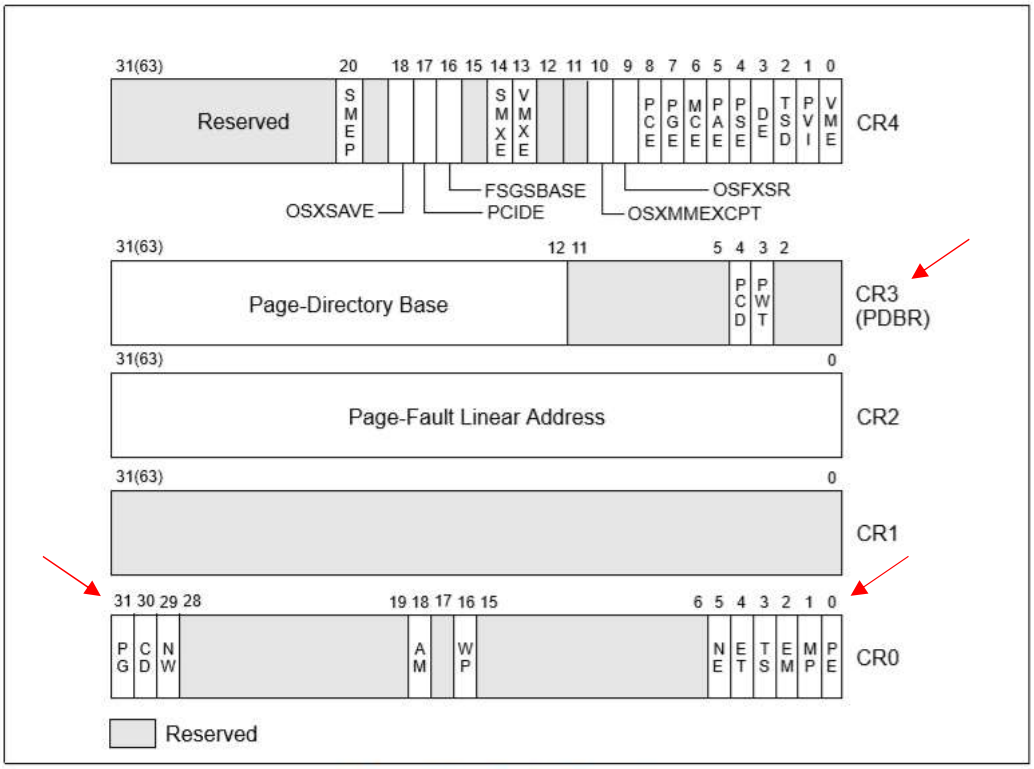
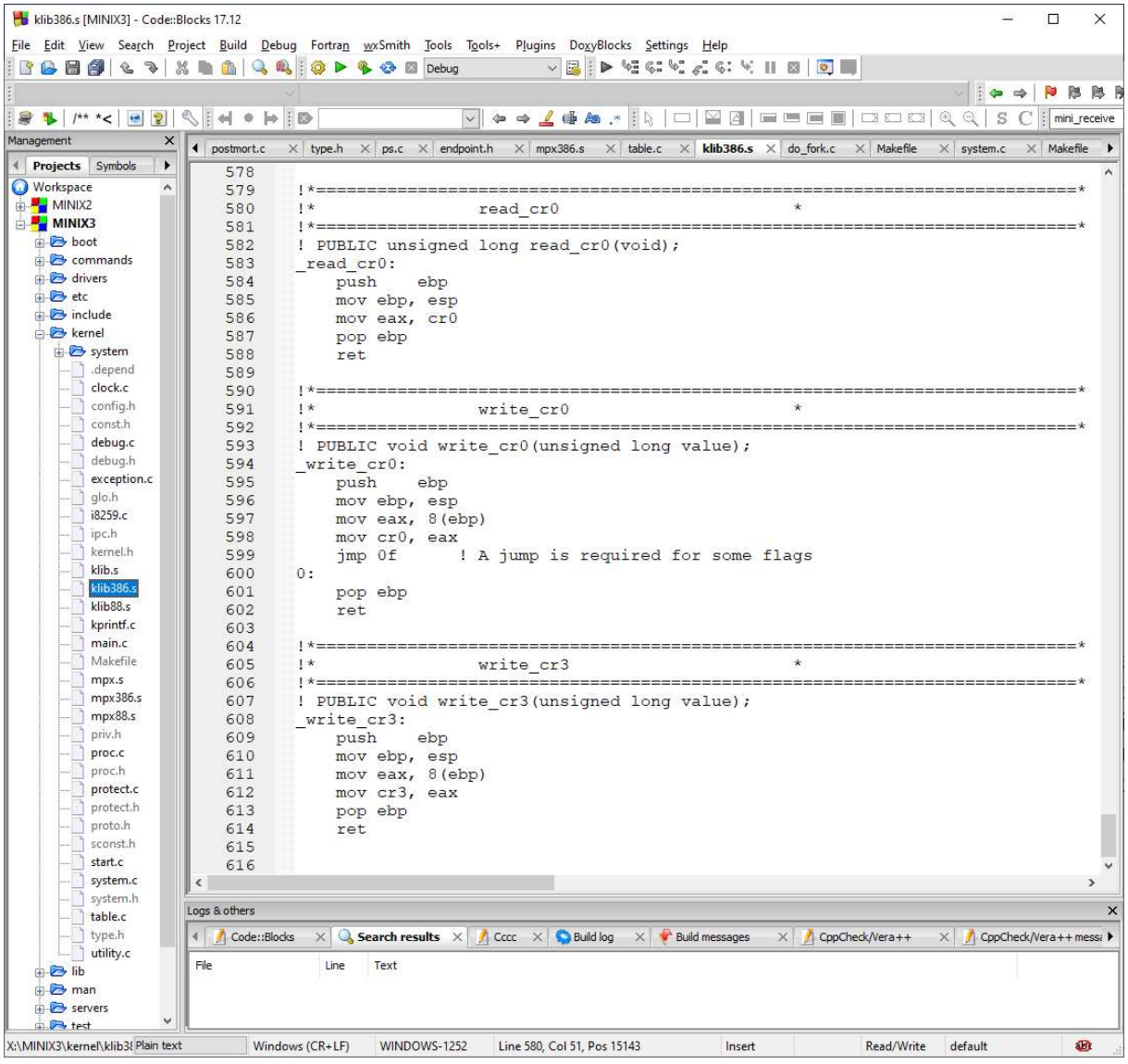


Figure 2-7. Control Registers

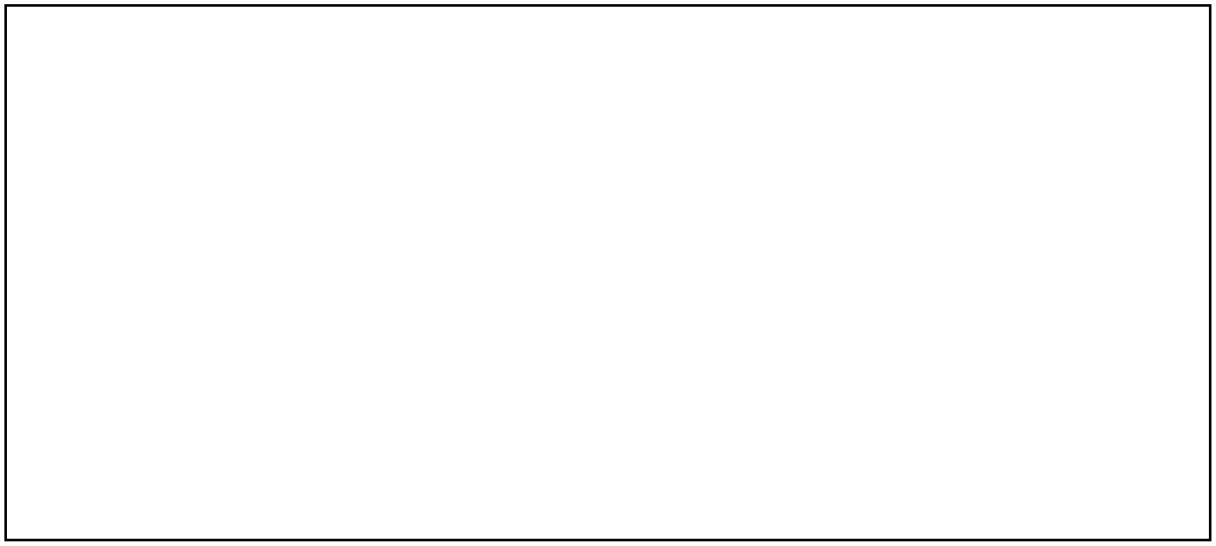
En resumen para activar la paginación, supuesto que la tabla de páginas está correctamente inicializada, necesitamos:

1) hacer que CR3 contenga la dirección de comienzo del directorio de páginas
2) poner a 1 el bit 31 de CR0 (bit PG)

La modificación de CR3 y CR0 sólo puede programarse en ensamblador ya que C como lenguaje de alto nivel no permite referenciar directamente registros hardware. Por suerte podemos evitar tener que programar en ensamblador las funciones necesarias para leer/escribir el contenido de CR3 y CR0, ya que en el fichero **/usr/src/kernel/klib386.s** (que está enlazado como parte del fichero ejecutable del núcleo, /usr/src/kernel/kernel) tenemos ya disponibles justo las funciones que nos hacen falta `read_cr0`, `write_cr0` y `write_cr3`:



- Indique en el recuadro siguiente cómo programaría en la función **main** de **/usr/sr/kernel/main.c** la activación de la paginación con la tabla de páginas a dos niveles que ha inicializado en el apartado anterior. Obviamente se deberá hacer uso de las funciones `read_cr0`, `write_cr0` y `write_cr3`.



5 RESOLVIENDO PROBLEMAS TÉCNICOS

Si en la función **main** de **/usr/src/kernel/main.c** se ha inicializado la tabla de páginas a dos niveles y se ha activado la paginación, es posible que tras recompilar el sistema y reiniciar Minix no termine de arrancar y se quede colgado. En este apartado vamos a indicar los problemas técnicos que pueden presentarse y que el alumno debe solucionar modificando convenientemente el código que ha añadido a la función **main**.

En primer lugar está claro que **main.c** debe incluir las cabeceras de las funciones funciones `read_cr0`, `write_cr0` y `write_cr3`, cosa que ya es así puesto que **main.c** incluye **kernel.h** y **kernel.h** incluye a su vez **proto.h** donde aparece ya explícitamente la declaración de los encabezamientos de esas tres funciones.

En segundo lugar el directorio de páginas y la tabla con las subtablas de página deben ser variables globales de **main.c** para que se ubiquen en el segmento de datos del núcleo, facilitándose así la conversión de las direcciones relativas a ese segmento expresadas mediante punteros de C, a direcciones físicas absolutas (como la requerida por CR3).

En tercer lugar el directorio de páginas (PD) debe cargarse en un marco, por lo que debe comenzar en una dirección múltiplo de 4 KB. Lo mismo para todas las subtablas de página (TP). Una forma de hacerlo sería declarar el directorio de páginas y las subtablas de páginas en un fichero en ensamblador y utilizar la directiva **.align 4096** en su declaración. Otra forma de resolver esto es utilizar punteros a las tablas en vez de variables globales, los cuales pueden inicializarse en direcciones múltiplo de 4 KB. Según eso declararíamos:

```
typedef unsigned descriptor_t ;                    /* 32 bits en Minix 3 */

descriptor_t * DP ;                  /* puntero al directorio de paginas */

descriptor_t * TP ;    /* puntero a la primera de las 8 tablas de paginas */

descriptor_t espacioTablas [ (1 + 8 + 1)* 1024 ] ;
```

Habría que inicializar los punteros DP y TP calculando en el programa los valores adecuados para que apuntaran a las tablas debidamente alineadas dentro del espacio reservado para almacenar la variable **espacioTablas**. Las asignaciones

```
DP = (descriptor_t *)&espacioTablas ;

TP = (descriptor_t *)((unsigned)DP + 4096) ;
```

Son un punto de partida, pero hay que mejorarlas para que ambos punteros queden apuntado a direcciones múltiplo de 4096 (el tamaño de los marcos/páginas).

En cuarto lugar las direcciones de memoria obtenidas con el operador & de C aplicado a variables globales proporcionan una dirección relativa al principio del segmento de datos y no una dirección física (absoluta). Por tanto es necesario conocer la dirección de comienzo del segmento de datos del núcleo para sumarla a esa dirección relativa. Dicha dirección base aparece bajo la columna **ds** (data segment) durante el arranque de Minix, tras el mensaje:

"Loading Boot image 3.1.2a revisión 4."

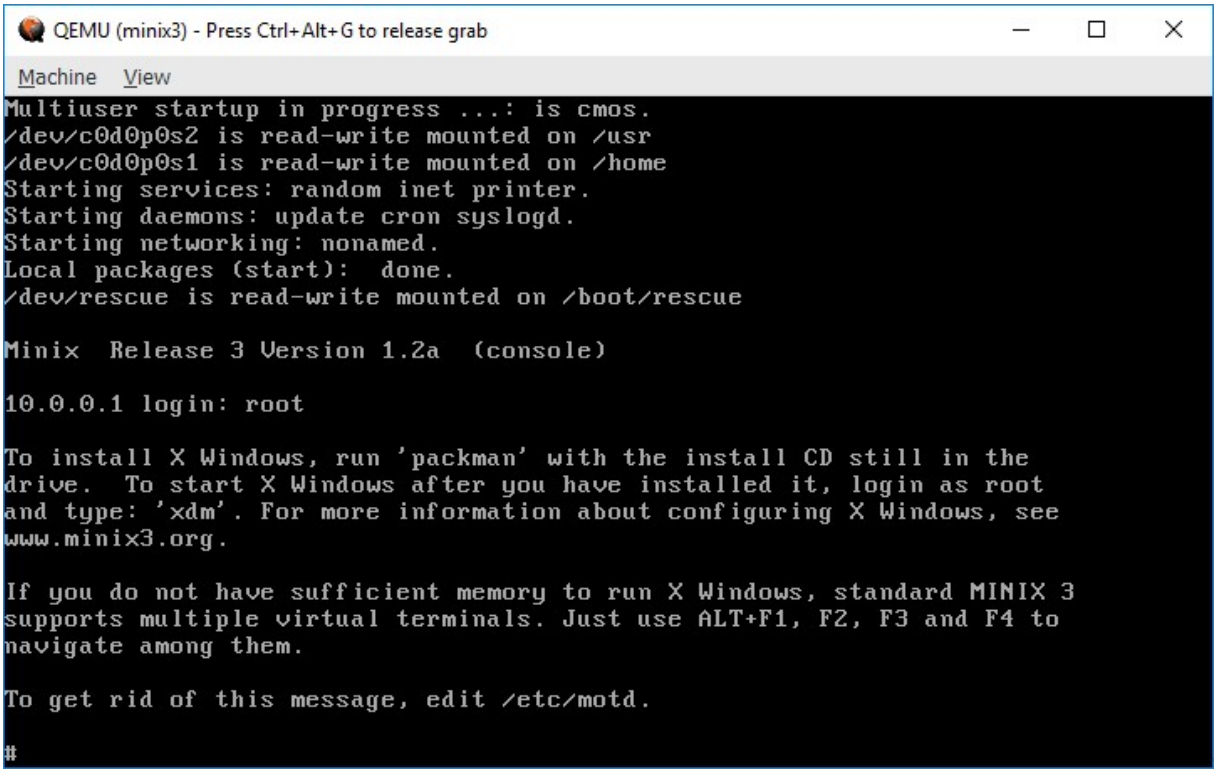
Por tanto podríamos tomar nota de la dirección de comienzo del segmento de datos del núcleo para sumarla a las direcciones relativas (como &DP[0] o &TP[0]) y obtener así las direcciones absolutas (como la que requiere el registro CR3). Sin embargo la dirección de comienzo del núcleo puede variar si posteriormente se añade código al núcleo, lo que tendría como consecuencia que la paginación nunca llegara a funcionar correctamente. Así, nos conviene utilizar la dirección contenida en `kinfo.data_base` y calculada en la función `cstart` de **/usr/src/kernel/start.c**. La estructura `kinfo` está declarada en **/usr/src/include/minix/type.h**, fichero que ya está incluido indirectamente en **main.c** a través de **kernel.h**.

Como método de depuración efectivo para comprobar que los registros CR0 y CR3 tienen el valor correcto en un punto concreto del código de **main.c**, puede ponerse en ese punto un bucle infinito (**for (; ;) ;**) y tras recompilar y reiniciar, ver el contenido de los registros con el comando info registers del monitor de qemu. Para ver el contenido en memoria de las tablas DP y PT podemos utilizar el comando **x** del monitor de qemu:

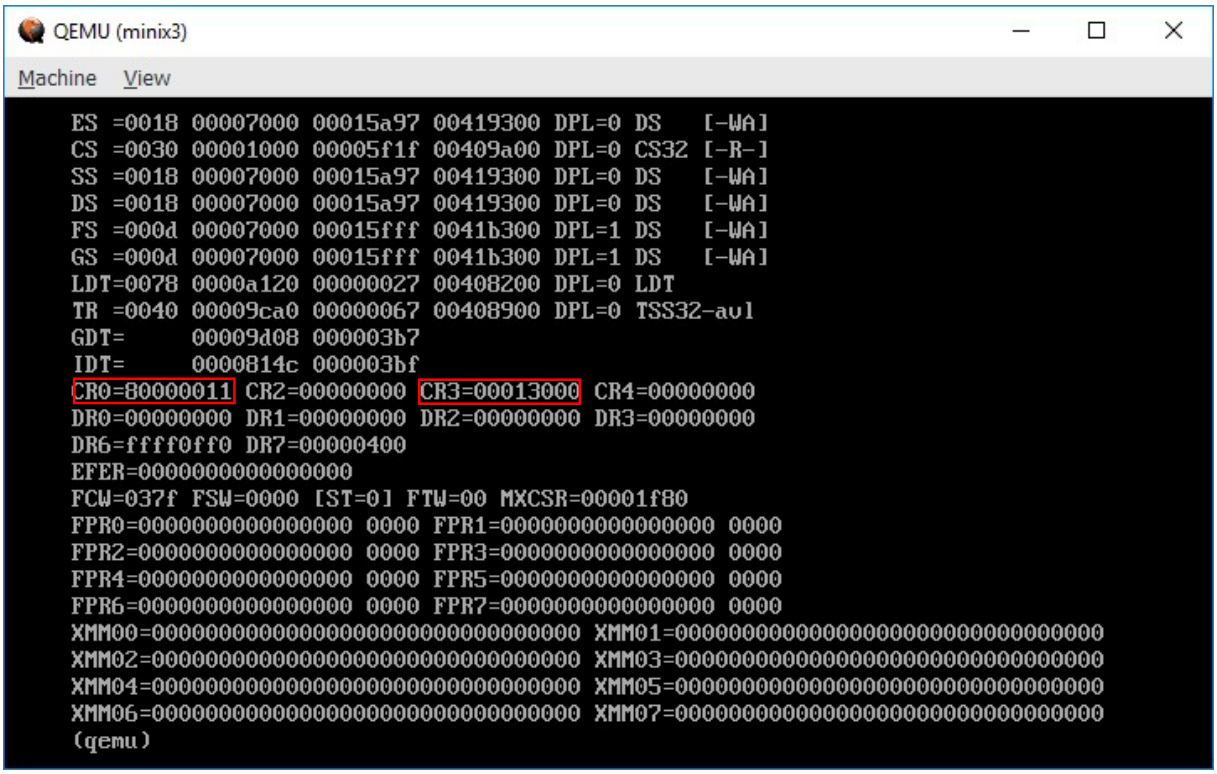
(qemu) **x /20x 0x00013000** y (qemu) **x /20x 0x00014000**

supuesto que 0x00013000 es la dirección física de comienzo de DP (RBDP) que debe estar establecido en el registro CR3, y supuesto que 0x00014000 es la dirección física de comienzo de PT.

Cuando la activación de la paginación se haya realizado correctamente Minix arrancará de manera normal y podremos abrir una sesión de root.



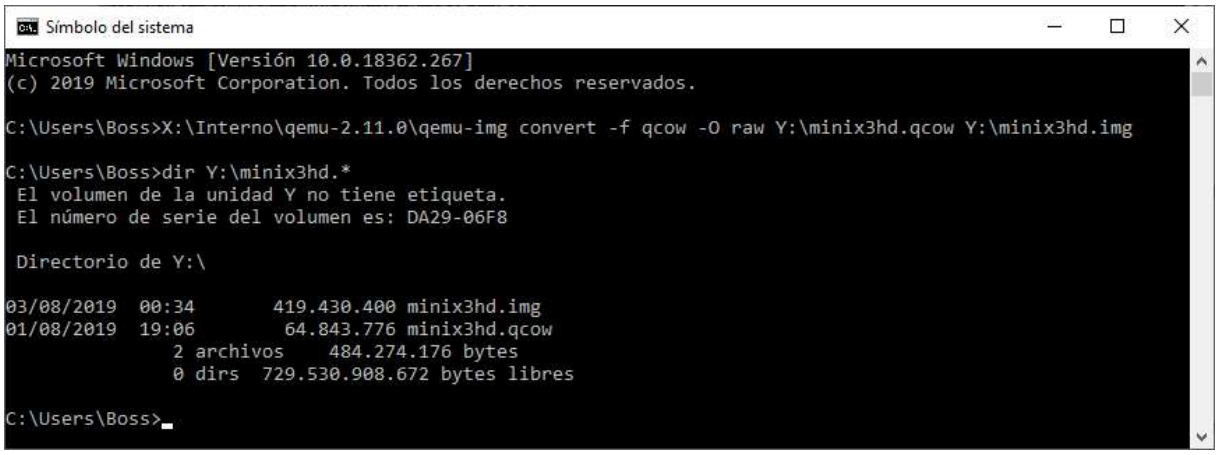
Si visualizamos el contenido de los registros con el comando info registers, debemos obtener una pantalla como la siguiente, con CR0 = 0x80000011 y CR3 = 00013000 (puede variar ya que dependerá del tamaño del código de **main.c**).



6 DEPURACIÓN CON BOCHS

La máquina virtual **Bochs** (http://bochs.sourceforge.net) dispone de un depurador mejorado con una interfaz gráfica muy potente que nos permite comprobar que la traducción de direcciones implementada es la que se solicita en esta práctica. El problema que hay es que Bochs sólo admite ficheros de imagen con formato **raw** y nuestra imagen **Y:\minix3hd.qcow** tiene un formato denominado **qcow** propio de qemu. Por tanto necesitaremos convertir la imagen de disco duro qcow: Y:\minix3hd.qcow a una imagen de disco duro raw que llamaremos **Y:\minix3hd.img**, la cual ya podremos arrancar con el depurador de Bochs.

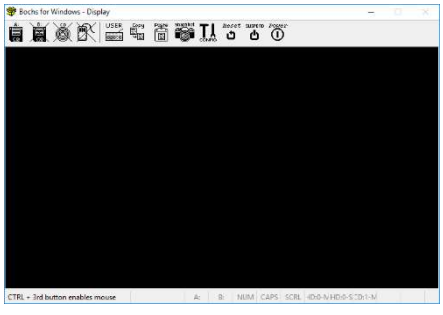
Para hacer la conversión qemu dispone del comando **qemu-img.exe** contenido en el directorio del software de prácticas **Y:\Interno\qemu-2.11.0**. Lo que tenemos que hacer es ejecutar **qemu-img.exe** en el intérprete de comandos de Windows como se indica en la pantalla siguiente: [o mas sencillo: pinchar en el escript X:\MINIX3\qcow2raw.cmd]

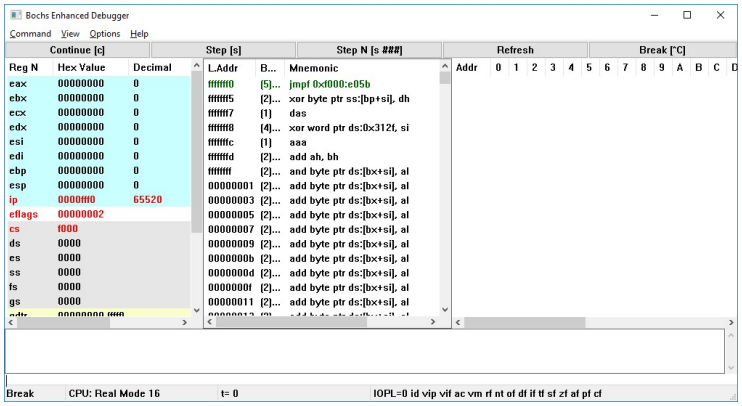


Una vez obtenida la imagen Y:\minix3hd.img lanzamos la máquina Bochs con el depurador gráfico pinchando en el icono de la aplicación:

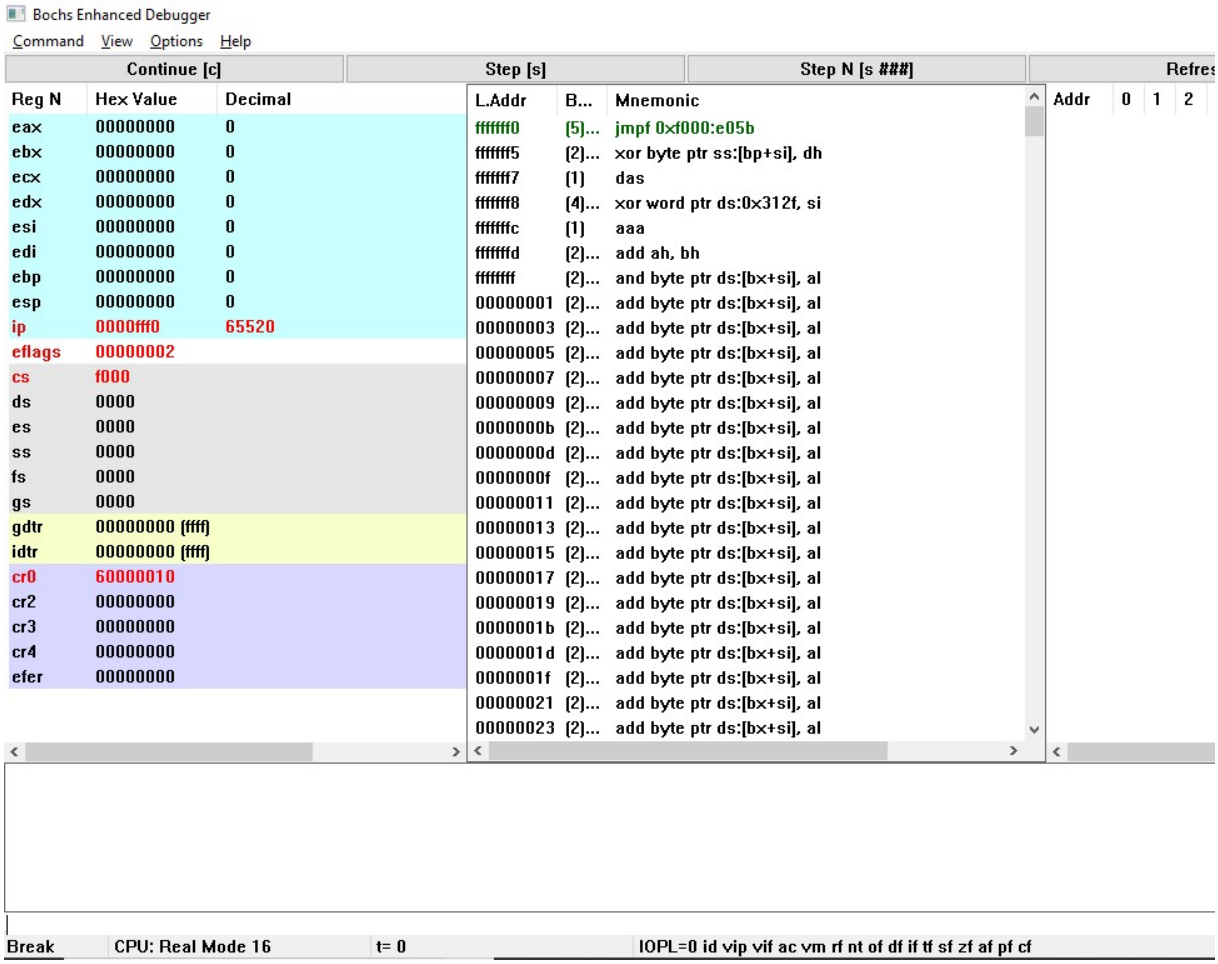
X:\MINIX3\minix3 bochsdbg (raw).exe

Como consecuencia aparecerán las ventanas de la máquina virtual Bochs y de su depurador de interfaz gráfica:

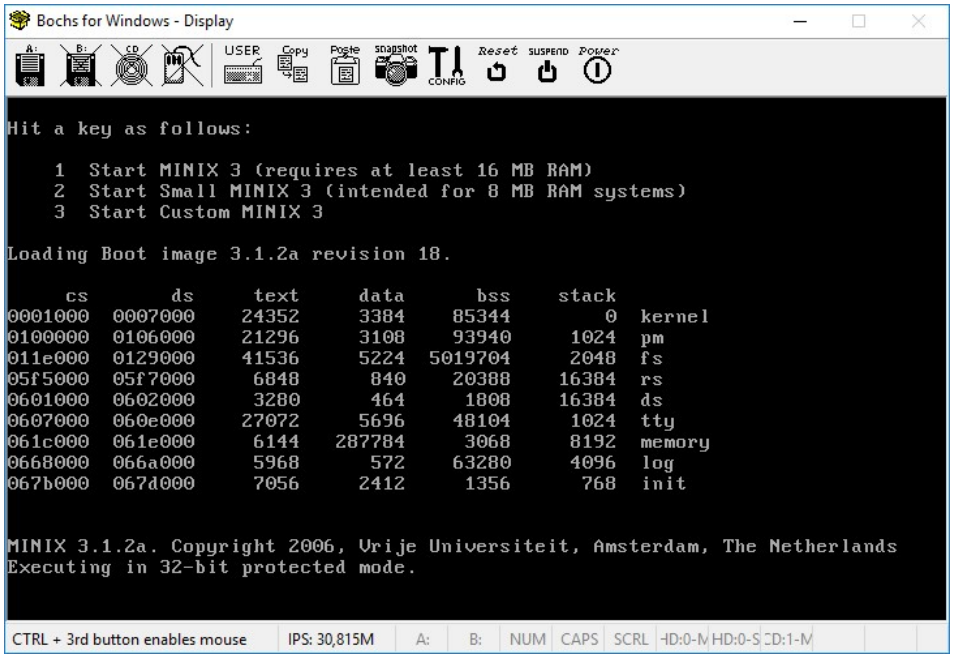




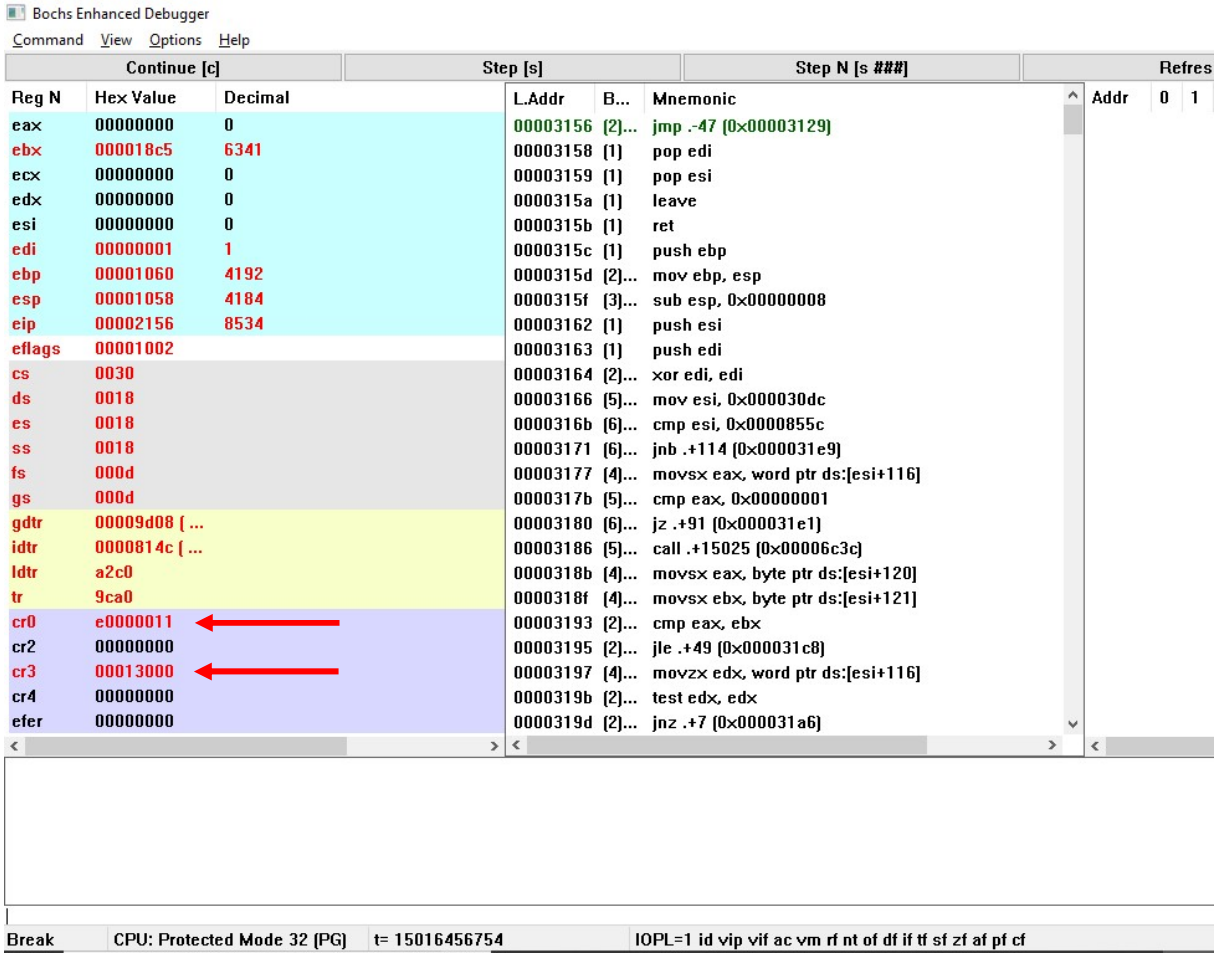
En un primer momento la máquina está parada justo en la primera instrucción que se ejecuta por parte del procesador al encender el ordenador. Para que comience a ejecutar instrucciones hay que pinchar sobre el botón **Continue [c]** del depurador.



Como puede observarse (CR0 = 0x60000010), inicialmente el procesador no está en modo protegido ni está activada la paginación. Pinchamos sobre el botón **Continue [c]** con lo que se ejecuta Minix en el depurador hasta que llega a un bucle infinito (**for (; ;) ;**) que se ha puesto en la función main tras la activación de la paginación.

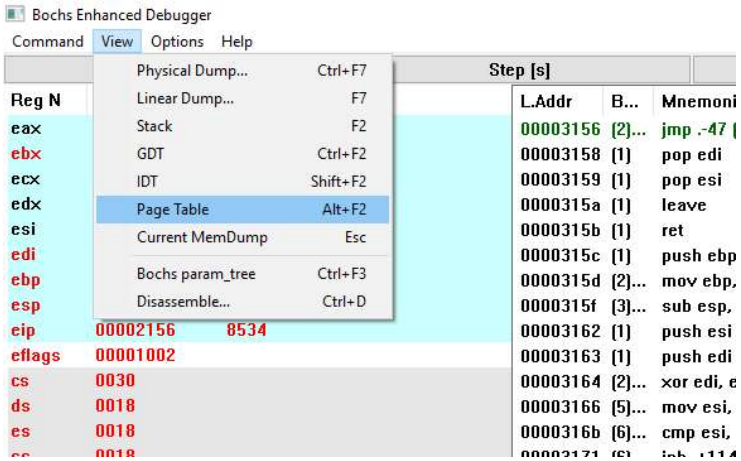


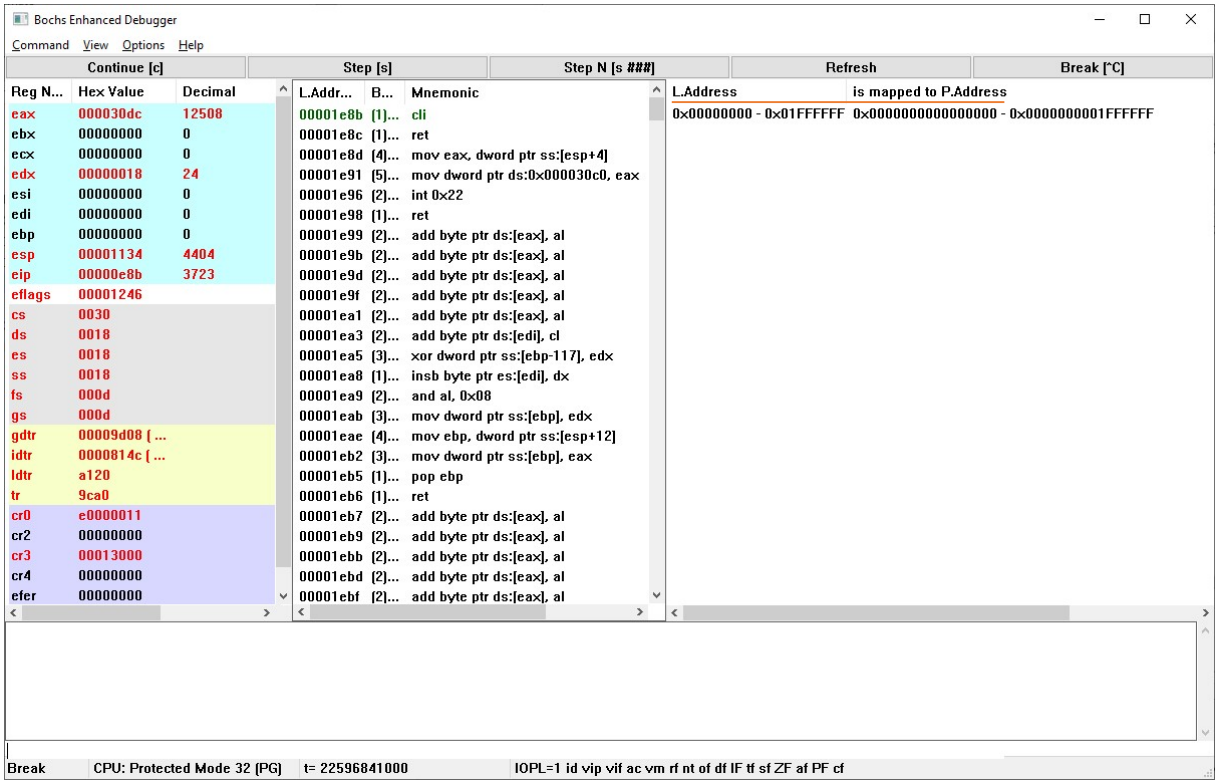
Pinchando sobre el botón **Break [^C]** del depurador (o tecleando **^C**) conseguimos detener la máquina Bochs para inspeccionar su estado en el depurador.



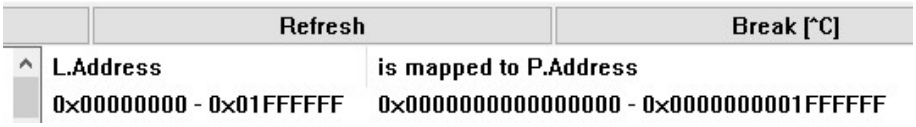
Podemos comprobar que estamos en modo protegido (**CR0 = 0xE0000011**) y con la paginación activada. Además el registro base del directorio de páginas (RBDP) contiene la dirección física 0x00013000 (**CR3 = 0x00013000**) lo que nos dice que el directorio de páginas está en esa dirección y por tanto en el marco 0x00013.

También podemos solicitar en el menú desplegable **View** visualizar de una manera lógica la tabla de páginas:

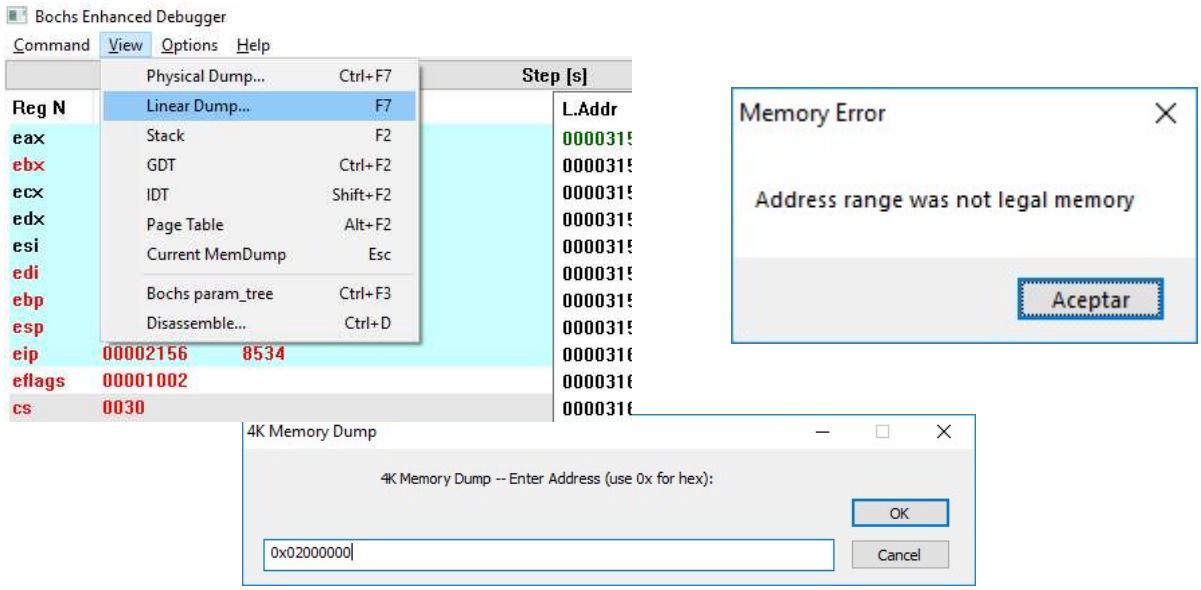




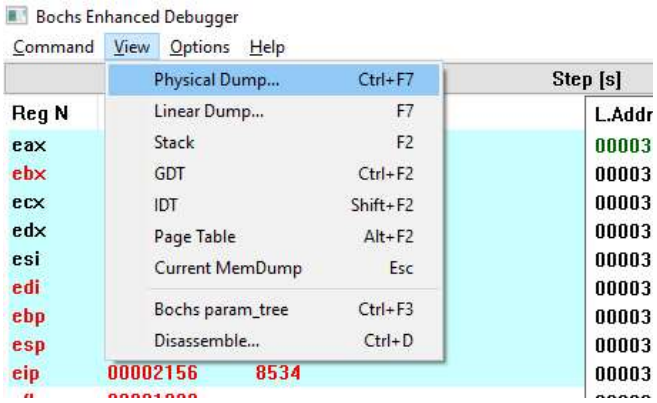
Vemos que a la derecha se nos dice que efectivamente las primeras 32 M direcciones virtuales se han mapeado (0 a 0, 1 a 1, … , 2^{25}-1 a 2^{25}-1) en las primeras 32 M direcciones físicas:

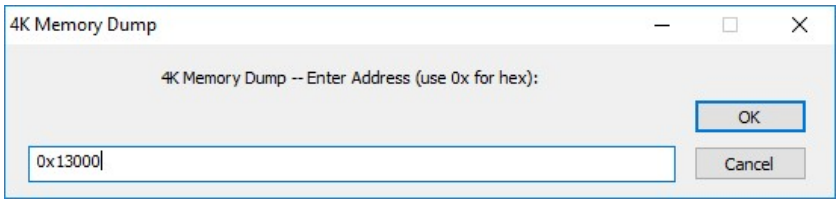


Además si intentamos acceder a la dirección virtual 32 M (0x02000000) vemos que dicha dirección no es accesible puesto que el descriptor correspondiente del directorio de páginas tiene el bit de presencia a 0. Eso no sucede con ninguna dirección dentro de los 32 MB.

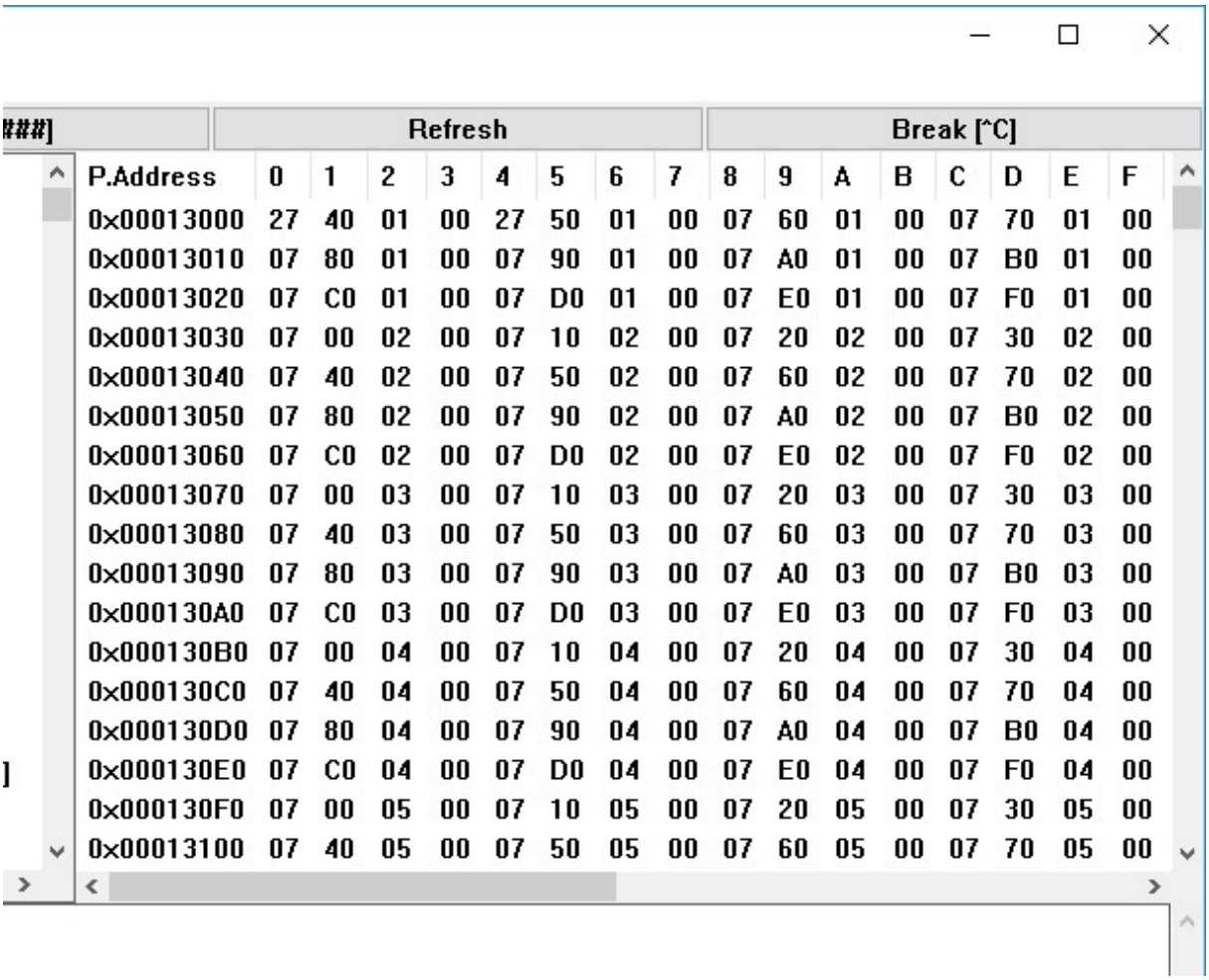


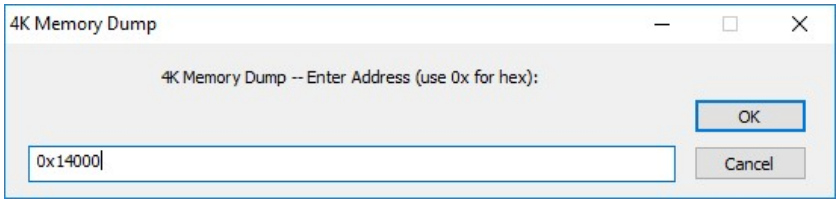
Por último podemos visualizar el contenido del directorio de páginas y de la primera subtabla de páginas haciendo un volcado físico a partir de la dirección 0x00013000 (directorio de páginas) y de 0x00014000 (primera subtabla de páginas) obteniendo lo siguiente:



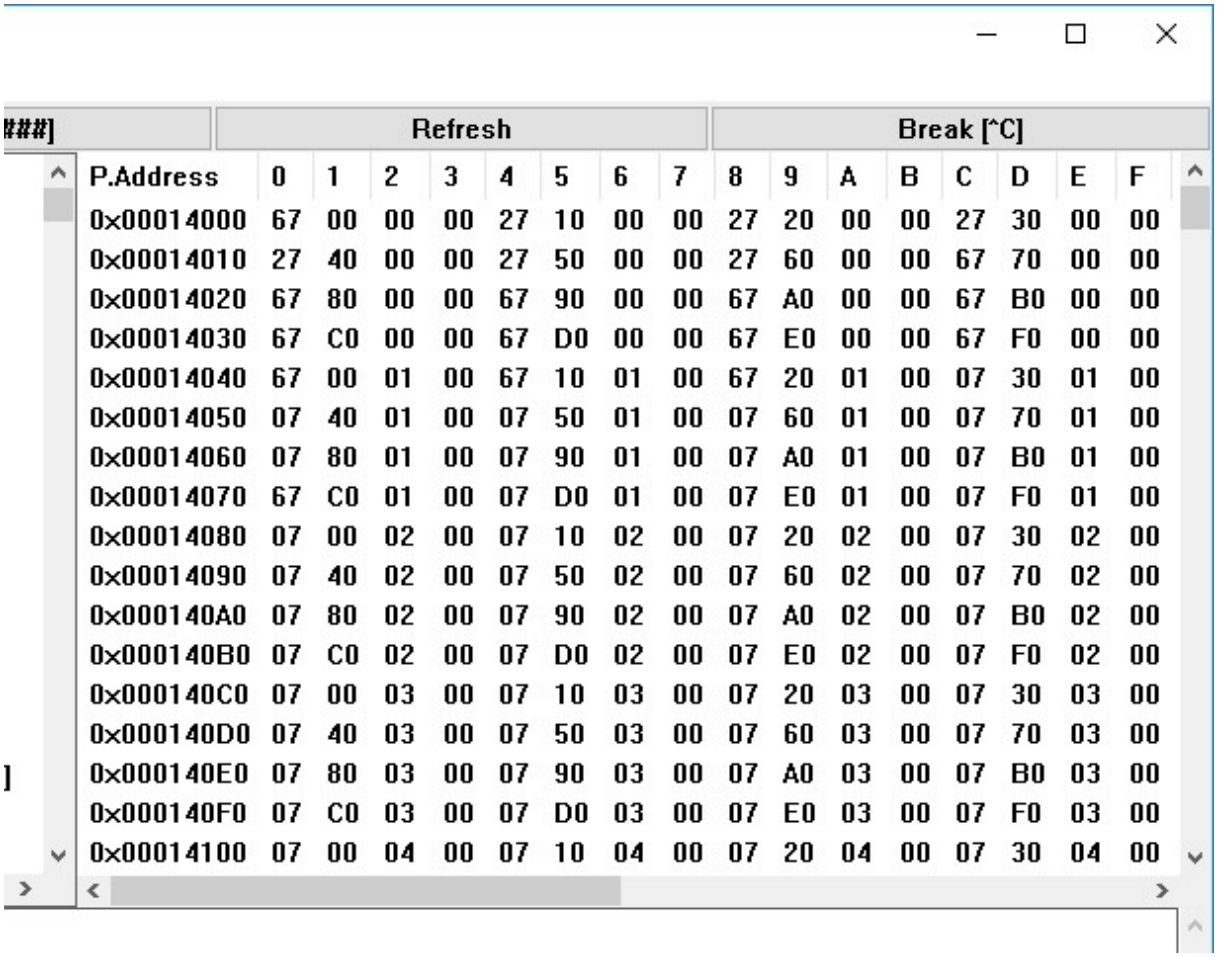


directorio de páginas:





primera subtabla de páginas



P.Address	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00014000	67	00	00	00	27	10	00	00	27	20	00	00	27	30	00	00
0x00014010	27	40	00	00	27	50	00	00	27	60	00	00	67	70	00	00
0x00014020	67	80	00	00	67	90	00	00	67	A0	00	00	67	B0	00	00
0x00014030	67	C0	00	00	67	D0	00	00	67	E0	00	00	67	F0	00	00
0x00014040	67	00	01	00	67	10	01	00	67	20	01	00	07	30	01	00
0x00014050	07	40	01	00	07	50	01	00	07	60	01	00	07	70	01	00
0x00014060	07	80	01	00	07	90	01	00	07	A0	01	00	07	B0	01	00
0x00014070	67	C0	01	00	07	D0	01	00	07	E0	01	00	07	F0	01	00
0x00014080	07	00	02	00	07	10	02	00	07	20	02	00	07	30	02	00
0x00014090	07	40	02	00	07	50	02	00	07	60	02	00	07	70	02	00
0x000140A0	07	80	02	00	07	90	02	00	07	A0	02	00	07	B0	02	00
0x000140B0	07	C0	02	00	07	D0	02	00	07	E0	02	00	07	F0	02	00
0x000140C0	07	00	03	00	07	10	03	00	07	20	03	00	07	30	03	00
0x000140D0	07	40	03	00	07	50	03	00	07	60	03	00	07	70	03	00
0x000140E0	07	80	03	00	07	90	03	00	07	A0	03	00	07	B0	03	00
0x000140F0	07	C0	03	00	07	D0	03	00	07	E0	03	00	07	F0	03	00
0x00014100	07	00	04	00	07	10	04	00	07	20	04	00	07	30	04	00

- Explique en el siguiente recuadro el contenido de las primeras entradas del **directorio de páginas** obtenidas anteriormente (ojo: el 80386 es *little endian*).

- Explique en el siguiente recuadro el contenido de las primeras entradas de la primera **subtabla de páginas** obtenidas anteriormente (ojo: el 80386 es *little endian*).

7 CAMBIANDO LA FUNCIÓN DE TRADUCCIÓN

Anteriormente el alumno ha implementado la función de traducción dv → df siguiente:

df = **si** (dv < 32 M)
dv [es decir FT(dv) = dv]
en otro caso
excepción (de falta de subtabla o de página) !!!!!

Ahora el alumno deberá modificar la implementación de la paginación para que la función de traducción dv → df sea esta otra:

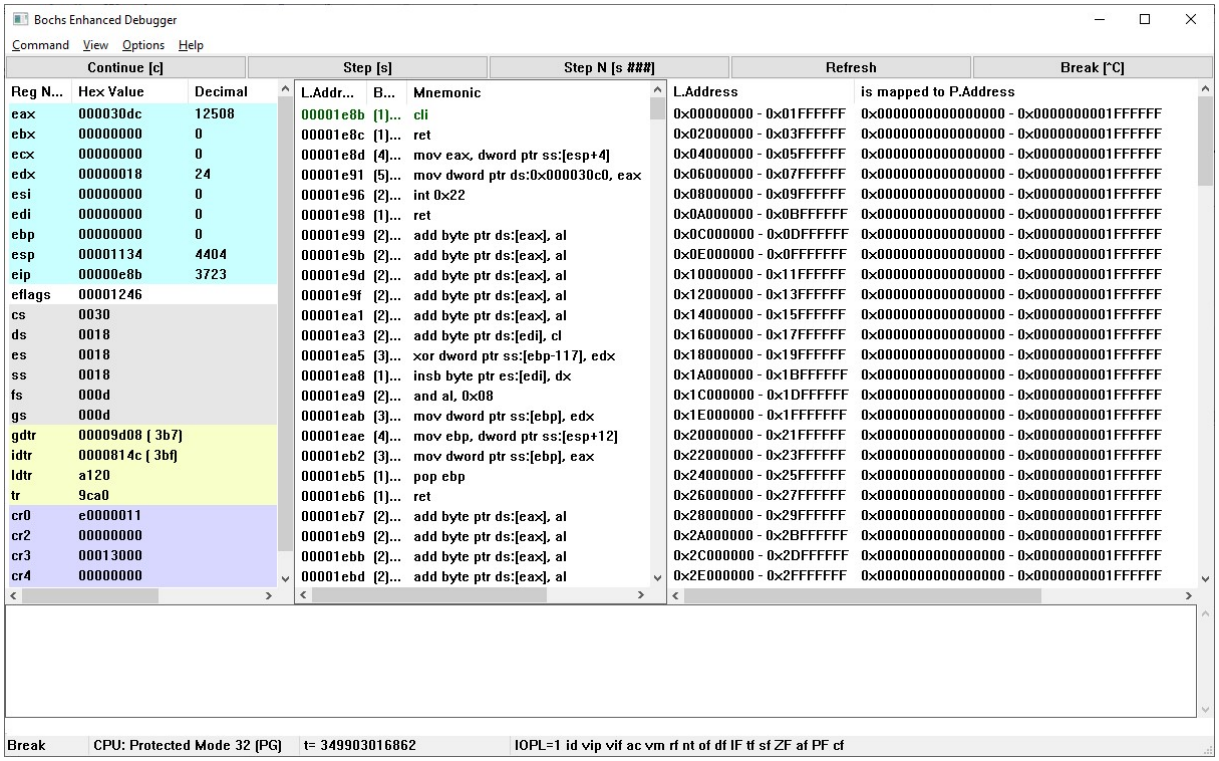
df = (dv % 0x2000000) [es decir FT(dv) = dv % 0x2000000]

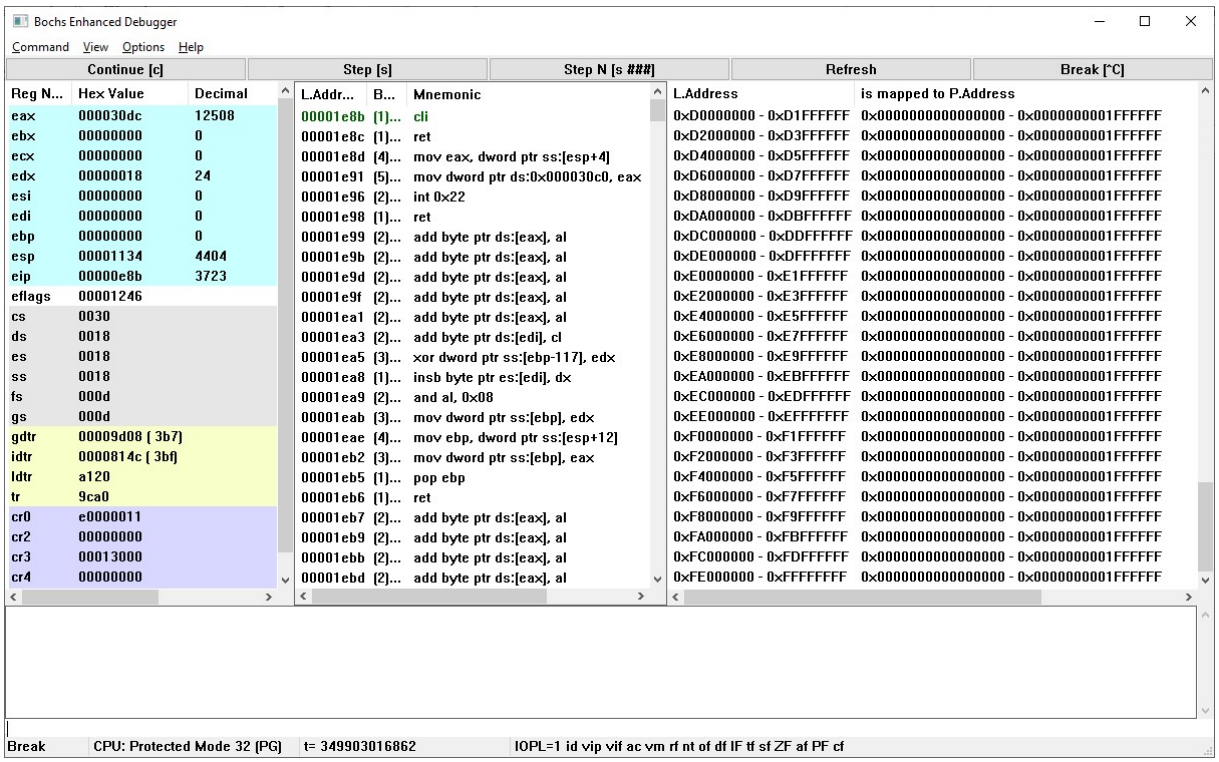
donde % representa el operador correspondiente al resto de la división entera (algoritmo de Euclides) y 0x2000000 es la representación en hexadecimal de 2^{25}, es decir 32 M.

- Programe la modificación de la inicialización de las tablas de paginación en **/usr/src/kernel/main.c** para que la función de traducción correspondiente a la función de traducción de direcciones sea la especificada anteriormente.

- Recompile Minix, arranque la máquina virtual qemu **X:\minix3.exe** y compruebe que se ha activado la paginación con la nueva función de traducción. Revise el contenido de los descriptores del directorio de páginas: PD[0], PD[8], PD[15], PD[16], PD[1016] y PD[1023]. Indique a continuación cual es el contenido de esos seis descritores (en hexadecimal).

- Arranque ahora la máquina virtual Bochs **X:\minix3 bochsdbg (raw).exe** y compruebe que se ha activado la paginación con la nueva función de traducción y muestre a continuación cuál es la tabla de páginas de acuerdo con el depurador de bochs (opción del menú **View-> Page Table**, Alt+F2). Explique si se corresponde con la función de traducción que se solicita en este apartado.


Bochs Enhanced Debugger
Command View Options Help
Continue [c]
Step [s]
Step N [s ###]
Refresh
Break [^C]
Reg N... Hex Value Decimal
eax 000030dc 12508
ebx 00000000 0
ecx 00000000 0
edx 00000018 24
esi 00000000 0
edi 00000000 0
ebp 00000000 0
esp 00001134 4404
eip 00000e8b 3723
eflags 00001246
cs 0030
ds 0018
es 0018
ss 0018
fs 000d
gs 000d
gdtr 00009d08 (3b7)
idtr 0000814c (3bf)
ldtr a120
tr 9ca0
cr0 e0000011
cr2 00000000
cr3 00013000
cr4 00000000
L.Addr... B... Mnemonic
00001e8b (1)... cli
00001e8c (1)... ret
00001e8d (4)... mov eax, dword ptr ss:[esp+4]
00001e91 (5)... mov dword ptr ds:0x000030c0, eax
00001e96 (2)... int 0x22
00001e98 (1)... ret
00001e99 (2)... add byte ptr ds:[eax], al
00001e9b (2)... add byte ptr ds:[eax], al
00001e9d (2)... add byte ptr ds:[eax], al
00001e9f (2)... add byte ptr ds:[eax], al
00001ea1 (2)... add byte ptr ds:[eax], al
00001ea3 (2)... add byte ptr ds:[edi], cl
00001ea5 (3)... xor dword ptr ss:[ebp-117], edx
00001ea8 (1)... insb byte ptr es:[edi], dx
00001ea9 (2)... and al, 0x08
00001eab (3)... mov dword ptr ss:[ebp], edx
00001eae (4)... mov ebp, dword ptr ss:[esp+12]
00001eb2 (3)... mov dword ptr ss:[ebp], eax
00001eb5 (1)... pop ebp
00001eb6 (1)... ret
00001eb7 (2)... add byte ptr ds:[eax], al
00001eb9 (2)... add byte ptr ds:[eax], al
00001ebb (2)... add byte ptr ds:[eax], al
00001ebd (2)... add byte ptr ds:[eax], al
L.Address is mapped to P.Address
0x00000000 - 0x01FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x02000000 - 0x03FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x04000000 - 0x05FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x06000000 - 0x07FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x08000000 - 0x09FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x0A000000 - 0x0BFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x0C000000 - 0x0DFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x0E000000 - 0x0FFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x10000000 - 0x11FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x12000000 - 0x13FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x14000000 - 0x15FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x16000000 - 0x17FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x18000000 - 0x19FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x1A000000 - 0x1BFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x1C000000 - 0x1DFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x1E000000 - 0x1FFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x20000000 - 0x21FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x22000000 - 0x23FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x24000000 - 0x25FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x26000000 - 0x27FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x28000000 - 0x29FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x2A000000 - 0x2BFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x2C000000 - 0x2DFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0x2E000000 - 0x2FFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
Break
CPU: Protected Mode 32 (PG)
t= 349903016862
IOPL=1 id vip vif ac vm rf nt of df IF tf sf ZF af PF cf


Bochs Enhanced Debugger
Command View Options Help
Continue [c]
Step [s]
Step N [s ###]
Refresh
Break [^C]
Reg N... Hex Value Decimal
eax 000030dc 12508
ebx 00000000 0
ecx 00000000 0
edx 00000018 24
esi 00000000 0
edi 00000000 0
ebp 00000000 0
esp 00001134 4404
eip 00000e8b 3723
eflags 00001246
cs 0030
ds 0018
es 0018
ss 0018
fs 000d
gs 000d
gdtr 00009d08 (3b7)
idtr 0000814c (3bf)
ldtr a120
tr 9ca0
cr0 e0000011
cr2 00000000
cr3 00013000
cr4 00000000
L.Addr... B... Mnemonic
00001e8b (1)... cli
00001e8c (1)... ret
00001e8d (4)... mov eax, dword ptr ss:[esp+4]
00001e91 (5)... mov dword ptr ds:0x000030c0, eax
00001e96 (2)... int 0x22
00001e98 (1)... ret
00001e99 (2)... add byte ptr ds:[eax], al
00001e9b (2)... add byte ptr ds:[eax], al
00001e9d (2)... add byte ptr ds:[eax], al
00001e9f (2)... add byte ptr ds:[eax], al
00001ea1 (2)... add byte ptr ds:[eax], al
00001ea3 (2)... add byte ptr ds:[edi], cl
00001ea5 (3)... xor dword ptr ss:[ebp-117], edx
00001ea8 (1)... insb byte ptr es:[edi], dx
00001ea9 (2)... and al, 0x08
00001eab (3)... mov dword ptr ss:[ebp], edx
00001eae (4)... mov ebp, dword ptr ss:[esp+12]
00001eb2 (3)... mov dword ptr ss:[ebp], eax
00001eb5 (1)... pop ebp
00001eb6 (1)... ret
00001eb7 (2)... add byte ptr ds:[eax], al
00001eb9 (2)... add byte ptr ds:[eax], al
00001ebb (2)... add byte ptr ds:[eax], al
00001ebd (2)... add byte ptr ds:[eax], al
L.Address is mapped to P.Address
0xD0000000 - 0xD1FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xD2000000 - 0xD3FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xD4000000 - 0xD5FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xD6000000 - 0xD7FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xD8000000 - 0xD9FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xDA000000 - 0xDBFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xDC000000 - 0xDDFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xDE000000 - 0xDFFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xE0000000 - 0xE1FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xE2000000 - 0xE3FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xE4000000 - 0xE5FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xE6000000 - 0xE7FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xE8000000 - 0xE9FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xEA000000 - 0xEBFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xEC000000 - 0xEDFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xEE000000 - 0xEFFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xF0000000 - 0xF1FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xF2000000 - 0xF3FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xF4000000 - 0xF5FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xF6000000 - 0xF7FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xF8000000 - 0xF9FFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xFA000000 - 0xFBFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xFC000000 - 0xFDFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
0xFE000000 - 0xFFFFFFFF 0x0000000000000000 - 0x0000000001FFFFFF
Break
CPU: Protected Mode 32 (PG)
t= 349903016862
IOPL=1 id vip vif ac vm rf nt of df IF tf sf ZF af PF cf

8 CONCLUSIONES

El alumno tras haber cumplimentado lo solicitado a lo largo de la práctica en una copia de este documento del word, deberá presentar el trabajo realizado al profesor aportando la máquina virtual qemu que ha utilizado y la imagen **minix3hd.qcow**, con el fin de ver que se ha establecido la paginación correctamente, y con el fin de inspeccionar el código desarrollado por el alumno especialmente en la función **main** del fichero **/usr/src/kernel/main.c**.

La conclusión que debe sacar el alumno es la naturaleza transparente de la paginación, ya que Minix 3.1.2a está escrito pensando que la memoria a la que se accede es física. Sin embargo al activar la paginación Minix 3.1.2a sigue funcionando sin ningún problema.

El alumno se preguntará ¿Qué hemos ganado al activar la paginación? La respuesta es que ahora tenemos nuevas posibilidades en el sistema operativo Minix. Por ejemplo podríamos añadir nuevas llamadas al sistema que permitieran compartir zonas de memoria entre los procesos sin mas que mapear diferentes páginas de los procesos sobre un mismo conjunto de marcos. Eso es algo que sin la paginación no podría llevarse a cabo.