PRACTICA 1. ARRANQUE DE Linux y COMPILACIÓN DEL kernel.

1.1. INTRODUCCIÓN

Las computadoras del laboratorio están configuradas con una máquina virtual VMware "Alumno" donde está instalado Ubuntu 9.04.

La entrada al sistema requiere un nombre de usuario (login) **alumno** y una palabra de paso (password) **dso2009**. Cada usuario entra al sistema con un intérprete de órdenes (shell) determinado y en un directorio concreto (home). En este caso, se debe entrar como superusuario o supervisor, "root", con el password "dso2009". El directorio *home* del superusuario es /root.

1.2. LAS ÓRDENES EN UNIX

A excepción de unas pocas órdenes propias del shell, la inmensa mayoría de las órdenes en UNIX son programas ejecutables. Los programas ejecutables se encuentran normalmente en los directorios /bin, /usr/bin, y alguno más. Realice la siguiente prueba:
echo $PATH

Con esta orden podrá ver los directorios en los que el shell busca los archivos ejecutables. Fíjese bien que el directorio de trabajo (.), por precaución, no suele estar en el path del superusuario.

El formato típico de una orden o programa en UNIX es: ***orden −opciones lista_argumentos***

- *orden* es el nombre del programa.
- *−opciones* es una lista de letras precedidas por el guión, cada opción modifica el funcionamiento del programa en uno u otro sentido.
- *lista_argumentos* son los elementos (archivos, procesos, etc.) sobre los que actuará la orden. Puede haber 0, 1 o más, separados por blancos (espacios, tabs, etc.). Recuerda que en UNIX siempre debe haber espacios de separación entre los componentes de la línea de órdenes. En MS−DOS, por ejemplo, podemos utilizar: *C:\> cd/dos* En UNIX eso significaría ejecutar el programa cd/dos, que probablemente no es lo que queremos. En su lugar, hemos de introducir un espacio entre el *cd* y su argumento: # *cd /dos*

1.3. AYUDA EN LÍNEA: ORDEN *man*

UNIX tiene integrada ayuda en línea en forma de "páginas del manual". El manual está dividido en volúmenes o secciones.

Para obtener las páginas de manual de una orden se usa la orden *man* [volumen] orden. Por ejemplo, pruebe # *man man* para ver las páginas de la orden *man*. Muchas de las páginas del manual están traducidas al castellano, si quieres ver el manual en castellano asigna a la variable de entorno LC_ALL el valor es_ES (español de España) mediante la orden: # export LC_ALL=es_ES

Para averiguar si existe una orden que realice una función concreta que no sabemos cómo llevar a cabo, puede buscarse por contenido en una base de datos de descripciones de las órdenes usando *man −k* (la k es de keyword). Para consultar las descripciones de una o más órdenes mediante su nombre, expresiones regulares o wildcards, puede usarse *whatis* o bien *man −f*. Prueba:
whatis mkdir
whatis −w mk*
whatis −r mk
apropos printer

Las páginas de manual de cada orden están en un archivo independiente que se llama orden.1, orden.2, etc., según el volumen del manual, y se almacenan en directorios como /usr/man/man1, /usr/man/man2, etc. Examine algunos directorios:
ls /usr/man
ls /usr/man/man1

Cada vez más, la información más completa va apareciendo en formato "info". Un método más intuitivo de consultar la ayuda en línea en sistemas con el interfaz gráfico GNOME es mediante el navegador de ayuda de gnome, gnome−help−browser. Por ejemplo, el manual *info* de la orden *find* se consultará así:
info find
o bien, con el navegador de gnome
gnome−help−browser info:find

En /usr/share/doc está la documentación de todos los paquetes. En /usr/share/doc/HOWTO están los archivos HOW−TO que son breves explicaciones sobre aspectos del sistema. En *Insflug* se traducen los "HOW−TO" al castellano.

1.4. MANEJO BÁSICO DE DIRECTORIOS Y ARCHIVOS

El sistema de archivos de UNIX es jerárquico, con un directorio raíz / en el que encontramos archivos y directorios. Dentro de estos últimos tenemos más archivos y directorios, y así sucesivamente. Jerarquía de directorios en UNIX:
Un directorio contiene siempre al menos dos elementos. Concretamente dos directorios:
. (punto) que apunta al propio directorio, y
.. (punto, punto) que apunta a su directorio padre. Los archivos que empiezan por punto se llaman "ocultos" y la orden *ls* no los muestra a no ser que indiquemos la opción −a (de all). Pruebe los siguientes ejemplos: {# ls −l} {# ls −la}

Para hacer referencia a un archivo utilizamos un path o pathname (que en castellano llamaremos ruta o, simplemente, nombre de archivo). Existen dos tipos:

- *nombres absolutos*, que empiezan desde el directorio raíz. Un archivo queda identificado siempre con el mismo nombre absoluto: {# ls −la /} {# ls −la /etc/passwd}
- *nombres relativos*, que empiezan desde el directorio actual. Dependiendo de en qué directorio nos encontremos, el archivo identificado puede variar. Podemos averiguar siempre el directorio actual con la orden *pwd*. {# cd /} {# ls −la .} {# cd /etc} {# ls −la passwd}. Ejemplos de algunos directorios:

Directorio	Descripción
/boot	Archivos relativos al arranque y al kernel.
/dev	Archivos especiales de dispositivo (discos periféricos, etc.)
/etc	Archivos de configuración del sistema.
/home	Directorios de los usuarios.
/lib	Librerías básicas y módulos del kernel.
/proc	Pseudodirectorio de acceso a datos del kernel y procesos.
/root	Directorio home del superusuario (root)
/tmp	Para el almacenamiento de archivos temporales.
/usr	Contiene todo lo demás (librerías, ejecutables, datos, etc.)
/var	Datos variables, spooling, logging, mail, etc.

Para cambiar de directorio se usa la orden *cd*, por ejemplo:

Orden	Descripción
cd cd ~	Nos llevan a nuestro directorio home, no importa donde estemos. (La vírgula representa abreviadamente a $HOME en el bash y se obtiene en el teclado español con AltGr−4.)
cd /	Nos lleva al directorio raíz.
cd ..	Nos lleva al directorio padre del actual.
cd .	No tiene efecto (nos lleva al directorio actual).
cd −	Nos lleva al último directorio en el que estuvimos antes de llegar al actual.

1.5. ÓRDENES COMUNES DE Linux

Del gran número de órdenes disponibles en un sistema UNIX, vamos a seleccionar las que más utilizaremos en estas prácticas.

- *ls*: Lista y muestra los atributos de archivos y directorios. Ejemplo: {# ls −lr /etc/i}
- *more*: Visualiza página a página un archivo. Con <return> se baja una línea, con espacio una página, y con q se sale antes del final. Ejemplo: {# more /etc/services}
- *mkdir*: Crea un directorio. Ejemplos: {# mkdir /home/alumno/mi_dir} {# mkdir −p mi_dir/otro_dir}
- *rmdir*: Borra un directorio vacío. Ejemplo: {# rmdir /home/alumno/mi_dir/otro_dir}
- *cp*: Copia archivos y directorios. Necesita al menos 2 parámetros. Ejemplos: {# cp /etc/i* .} {# cp −r /etc/init.d .}
- *mv*: Cambia de nombre o mueve a otro directorio un archivo. Ejemplos: {# mv k1 k99} {# mv /home/alumno/k1 /home/alumno/mi_dir}
- *rm*: Borra archivos y directorios. Ejemplos: {# rm /home/alumno/k1} {# rm −r /home/alumno/mi_dir}
- *ln*: Crea un enlace a un archivo. El enlace puede ser duro/fuerte o simbólico. Ejemplos: {# ln fich_existente nuevo_fich} {# ln −s fich_existente nuevo_fich}
- *diff*: Compara dos archivos. Ejemplo: {# diff prog1.cprog2.c}
- *grep*: Busca patrones en archivos. Ejemplo: {# grep www /etc/services}
- *locate*: Búsqueda básica de archivos en todo el sistema. Ejemplos: {# locate profile} {# locate limit}
- *which*: Nos dice qué archivos se ejecutarían si escribiéramos sus argumentos en el actual shell. Ejemplos: {# which ls} {# which which}
- *gzip*: Comprime archivos. Ejemplos: {# cp /usr/share/dict/spanish . ; gzip spanish}
- *gunzip*: Descomprime lo que comprime gzip.
- *bzip2*, *bunzip2*: Un compresor normalmente más eficaz.
- *tar*: Empaqueta y desempaqueta archivos. Ejemplos: {# tar −czvf copia.tgz /home/alumno} {# tar −tzvf copia.tgz} {# tar −xzvf copia.tgz}

1.6. EL ARRANQUE DEL SISTEMA

Los sistemas del laboratorio están configurados con varias instalaciones separadas de Linux. Al arrancar la máquina y cargarse GRUB desde el MBR del disco duro, el cargador da opción de seleccionar diferentes versiones de FEDORA (5 y 6). El bootmanager solicita un usrname y una password, que es “root” y “fedora5” o “fedora6”. Una vez seleccionado, GRUB carga el sector de arranque de la partición donde está instalado el sistema Linux para la asignatura. Todas las modificaciones que hagamos al arranque afectarán al GRUB. El archivo de configuración de GRUB es /boot/grub/grub.conf, que posteriormente se modificará para añadir la opción de arrancar nuestro kernel. Es necesario recordar que SIEMPRE han de cerrarse los ordenadores con la orden shutdown –h now o pulsar Ctrl+Alt+Supr, a riesgo de que el sistema de ficheros quede deteriorado.

Dado el carácter de la asignatura, es necesario que todos los alumnos tengan acceso como administrador o root al sistema. Este usuario puede ejecutar cualquier comando privilegiado mediante la utilización de la orden sudo, que solicita al usuario su password antes de realizar la acción solicitada. Ni que decir tiene que este“privilegio” no es excusa para instalar en el sistema ningún tipo de servidor o programa no autorizado, o con fines malígnos.

La sesión se inicia automáticamente en el entorno X-Window. Si desea trabajar exclusivamente en modo consola, puede pulsar Ctrl+Alt+Fn, siendo n un número entre 1 y 4. Después puede conmutar entre consolas con Alt+Fn, o bien volver a Xwindow pulsando Ctrl+Alt+F7.

Tras la inicialización básica por parte de la BIOS, y siendo el arranque desde disco duro, el sistema carga el sector de MBR en la memoria RAM y ejecuta el programa que allí se encuentra. En nuestro caso, el programa que se encuentra (parcialmente) en el MBR es GRUB, el cual nos permite seleccionar qué sistema operativo iniciar.

En nuestro caso, el GRUB con el que nos vamos a enfrentar es el que reside en el primer sector de la partición donde está el sistema Linux de la asignatura, pero a todos los efectos esto no es relevante. Para el desarrollo de la asignatura no es necesario conocer a fondo el funcionamiento de GRUB; baste decir que GRUB es capaz de leer del sistema de archivos y actuar en función del contenido del archivo /boot/grub/grub.conf, por lo que sólo necesitaremos modificar este archivo para trabajar.

Del archivo, la parte más interesante es la que se refiere a las opciones del menú de arranque. Cada entrada tiene varios comandos de GRUB que se ejecutan consecutivamente si la opción se selecciona, como por ejemplo:

- title: Título de la opción que aparecerá en el menú de arranque.
- root: Partición que contiene el kernel a cargar indicado en la línea kernel.
- kernel: Ruta del archivo que contiene el kernel a cargar, más los parámetros que se le pasarán al kernel. La ruta es dentro de la partición indicada en el parámetro root.
- initrd: Imagen initrd relacionada con el kernel seleccionado, en su caso.
- savedefault: Indica que si esta opción se selecciona, se guarda como opción por defecto para el siguiente arranque.
- boot: Da la orden de arrancar a GRUB con los parámetros configurados en el momento.

1.7. UTILIDADES PARA LA COMPILACIÓN Y ENLAZADO (gcc y make)

El compilador que utilizaremos en prácticas será el *gcc* (GNU C Compiler). Como todo sistema operativo de tipo UNIX, Linux está implementado en C, y *gcc* está considerado como una de los mejores productos GNU existentes y en general es uno de los más potentes. La orden *gcc* permite encadenar las diferentes etapas de compilación de un programa (preprocesado, compilado, ensamblado y enlazado). Es decir, *gcc* es un programa que ejecuta las diferentes etapas, transmitiendo a los programas las opciones proporcionadas por el programador.

Opciones interesantes de *gcc* podrían ser: -v (visualiza los diferentes programas lanzados por gcc); -M (parada tras la etapa del preprocesador (cpp)); -S (para tras la generación del código ensamblador (cc1)); -c (parada tras la generación del código objeto (as)); -o (parada tras la generación de código ejecutable (ld)); -g (generación de los símbolos para depuración); **-o**nomdest (nombre del archivo generado); **-D**MACRO (define la macroinstrucción); **-D**MACRO=AVALOR (define la macroinstrucción y le asigna un valor); **-I**DIRECTORIO (directorio donde *cpp* (preprocesador) debe ir a buscar los archivos cabecera); **-L**DIRECTORIO (directorio donde *ld* (enlazador) debe ir a buscar las librerías); **-l**libreria (incluye la librería para el enlazado); -pipe (encadenamiento de las diferentes opciones de compilación sin utilizar archivos temporales, aunque esta opción acelera el proceso de compilado requiere más memoria); **-O**n (nivel de optimización, n va desde 0 hasta 3).

Indicar que el número de opciones de *gcc* es enorme, para ello es recomendable consultar la página del manual relativa a *gcc*.

La herramienta *make* facilita la compilación de proyectos, y es una herramienta estándar instalada en todo sistema UNIX. Este programa permite efectuar una compilación inteligente de programas, en función de los archivos modificados que necesitan realmente ser compilados. No obstante, la sintaxis de los archivos *Makefile* es relativamente compleja de escribir porque utiliza reglas de reescritura, aunque aquí no las veremos todas en detalle.

Al ser ejecutado, *make* utiliza el archivo *makefile* o *Makefile* situado en el directorio actual, analiza su contenido y lanza las ordenes indicadas. Esta herramienta es muy práctica porque permite, entre otras cosas, recompilar sólo lo necesario.

El ejemplo que ilustra este apartado se puede utilizar también en los ejercicios relacionados con compilación de proyectos en estas prácticas. Este ejemplo usa varios archivos: *complejo.c*: módulo que implementa los cálculos sobre números complejos; c*omplejo.h:* interfaz de prototipos del módulo *complejo.c*; *dibujoc.c:* módulo de representación gráfica de los números complejos; *dibujo.h:* interfaz de los prototipos del módulo *dibujo.c; main.c:* archivo principal. El archivo *Makefile* asociado es entonces el siguiente:

```
# Ejemplo de archivo makefile para números complejos
CC=gcc
RM=/bin/rm
# Opciones de compilación
CFLAGS:-g
# Archivos
SRC=complejo.c dibujo.c main.c
OBJ=$(SRC: .c=.o)
PROGRAMA=complejo
LIB=-lm
# Reglas de generación
$(PROGRAMA) : $(OBJ)
$(CC) $ (CFLAGS) $(OBJ) -o $(PROGRAMA) $(LIB)
# Propiedad
clean:
$(RM) -f $(CBJ) $ (PROGRAMA)
# Dependencias
complejo.o      : complejo.c complejo.h
dibujo.o        : dibujo.c dibujo.h complejo.h
main.c          : main.c dibujo.h complejo.h
```

La primera parte de este archivo está constituida por declaraciones diversas que permiten una cierta flexibilidad de uso en el caso en que se quiera, por ejemplo, cambiar de compilador. Seguidamente, se definen las opciones que deberán utilizarse en la compilación. Aquí se trata de la opción *-g* para poder depurar eventuales errores.

Viene a continuación la enumeración de los distintos archivos que deben compilarse. La variable OBJ está formada por una regla que evita tener que enumerar todos los archivos objeto sabiendo que se trata de los mismos nombres que para los archivos fuente, aparte de la extensión.

La última parte es la regla de creación del ejecutable. Esta regla se ejecuta únicamente si todos los archivos objeto han sido generados. Al fin del archivo se encuentran las dependencias que permitirán a *make* compilar sólo los archivos modificados.

Por ejemplo, supongamos que se hayan generado todos los archivos objeto, pero desde la última compilación un desarrollador ha modificado el archivo *dibujo.h.* En la próxima compilación, sólo se recompilarán ciertos archivos:

```
# make
gcc -g -c dibujo.c -o dibujo.o
gcc -g -c main.c -o main.o
gcc -g complejo.o dibujo.o main.o -o complejo -lm
```

Gracias a la definición de dependencias, el archivo *complejo.c* no se recompila. Para conocer las dependencias de un programa, basta con lanzar la orden *gcc -MM dibujo.c*. Se obtiene así la lista de archivos de cabecera utilizados por este módulo.

1.8. COMPILACIÓN DEL KERNEL

Para compilar el núcleo lo primero es conseguir el código fuente. El código fuente de Linux se puede descargar de http://www.kernel.org o cualquiera de sus mirrors. Los fuentes están en un archivo comprimido con extensión gz, bz o tgz, que se descomprimen con las herramientas correspondientes. En nuestro caso tenemos Ubuntu, y la compilación del kernel se realizará de la siguiente forma:

Se obtienen los paquetes necesarios para construir el kernel:
>sudo apt-get install build-essential kernel-package libncurses5-dev
Se descarga el source del kernel desde los repositorios de la distribucion
>sudo apt-get install linux-source
Descomprimir el fichero comprimido en /usr/src
>sudo tar jxvf linux-source-2.6.28.tar.bz2
Se copia la configuracion antigua de /boot
>cd linux-source-2.6.28/
>sudo cp /boot/config-2.6.28-15-generic .config
Se compila el nuevo kernel
Si se quiere elimianar algun modulo o funcionalidad antes se deben elegir las opciones:
>sudo make menuconfig
Si no directamente se compila el nuevo kernel:
>sudo make-kpkg clean
>sudo make-kpkg --initrd --append-to-version=-dso1 kernel_image kernel_headers

Instalacion del nuevo kernel, se instalan los paquetes generados en /usr/src

>sudo dpkg -i linux-image-2.6.28.10-dso1_2.6.28.10-dso1-10.00.Custom_i386.deb
>sudo dpkg -i linux-headers-2.6.28.10-dso1_2.6.28.10-dso1-10.00.Custom_i386.deb

La instalacion ya actualiza el grub por lo que solo sera necesario reiniciar para poder usar el nuevo kernel.

1.9. PROGRAMAS EJEMPLO (PROYECCIÓN EN MEMORIA DE ARCHIVOS)

Linux proporciona varias llamadas al sistema que permiten *proyectar* el contenido de archivos en memoria (__ptr_t es equivalente a void *):

```
#include <unistd.h>
#include <sys/nwan.h>
extern __ptr_t mmap(__ptr_t start, size_t len, int prot, int flags, int Fd, __off_t offset);
extern int munmap(__ptr_t start, size_t len);
extern __ptr_t mremap(__ptr_t *old_start, size_t old_len, size_t new_len, unsigned long flags);
extern int mprotect(__ptr_t start, size_t len, int prot);
```

La función de biblioteca *mmap* proyecta el contenido de un archivo en memoria, en el espacio de direccionamiento del proceso actual. El parámetro *start* especifica la dirección de la zona de memoria donde el contenido del archivo debe ser accesible. Esta dirección es únicamente una indicación dada al sistema, que puede decidir utilizar otra de inicio. El parámetro *len* indica el número de bytes a proyectar en memoria. Las protecciones en memoria a aplicar se definen por el parámetro *prot*, mediante las siguientes macros: PROT_NONE (la zona se marca como inaccesible), PROT_READ (la zona se marca como accesible en lectura), PROT_WRITE (la zona se marca como accesible en escritura) y PROT_EXEC (la zona se marca como accesible en ejecución). El parámetro *flag* especifica las modalidades de la protección. Finalmente, *fd* indica el descriptor del archivo, cuyo contenido debe proyectarse en memoria, y *offset* especifica a partir de qué byte el contenido del archivo debe ser accesible. La dirección a partir de la cual el contenido del archivo es accesible se devuelve por *mmap.* En caso de error, *mmap* devuelve el valor -l.

Pueden utilizarse varias constantes para el parámetro *flag*, definidas en el archivo <sys/rnrr,an.h>: (1) MAP_FIXED (La dirección de inicio debe imperativamente corresponder al parámetro addr. Si esta dirección no se puede usar, *mmap* devuelve un error); (2) MAP_SHARED (La proyección en memoria se comparte con todos los otros procesos que hayan proyectado el archivo en memoria. Toda modificación efectuada por un proceso es inmediatamente visible por los otro); (3) MAP_PRIVATE (La proyección en memoria sólo afecta al proceso actual. Toda modificación efectuada por el proceso actual no será visible por los otros procesos que han proyectado el archivo en memoria); (4) MAP_ANONYMOUS (La proyección en memoria no afecta a ningún archivo. La primitiva *mmap* se llama para crear una nueva región de memoria cuyo contenido se inicializará a cero); (5) MAP_GROWSDOWN (La zona de memoria se orienta o crece hacia abajo (stack)); (6) MAP_DENYWRITE (Todo intento de acceso en escritura al archivo por un proceso devolverá el error ETXBSY); (7) MAP_EXECUTABLE (La zona de memoria proyectada o mapeada se marca como un biblioteca o librería); (8) MAP_LOCKED (La zona de memoria creada debe bloquearse en memoria); y (9) MAP_NORESERVE (después de la proyección, no se comprueba si hay suficiente memoria libre).

En caso de error, la variable errno puede tomar los valores siguientes: (1) EACCES (El tipo de proyección o las proyecciones de acceso son incompatibles con el modo de apertura del archivo); (2) EAGAIN (El archivo está bloqueado o una cantidad demasiado importante de páginas están bloqueadas en memoria); (3) EBADF (El descriptor de entradas/salidas especificado no es válido); (4) EINVAL (*start*, *len* u *offset* contiene un valor no válido (por ejemplo una dirección que no está alineada a una frontera de página)); (5) ENOMEM (No existe bastante memoria disponible); (6) ETITBSY (La opción MAP_DENYWRITE se ha especificado pero el archivo está abierto en escritura).

La primitiva *munmap* suprime la proyección en memoria de un archivo. El parámetro *start* especifica la dirección de la zona de memoria correspondiente y *len* indica su tamaño, expresado en bytes. En caso de éxito, se devuelve el valor 0. En caso de error, *munmap* devuelve el valor -1, y la variable errno toma el valor EINVAL.

La llamada al sistema *mremap* modifica el tamaño de una zona de memoria. El parámetro *old_start* especifica la dirección de inicio de la zona, cuyo tamaño en bytes se indica en *old_len*. El nuevo tamaño de la zona se transmite en el parámetro *new_len*. El parámetro *flags* especifica las modalidades de la modificación. Linux 2.0 sólo proporciona una opción, MREMAP_MAYMOVE, que indica que el kernel está

autorizado para modificar la dirección de inicio de la zona. La primitiva *mremap* devuelve la dirección de la zona de memoria, que puede ser diferente del valor transmitido en *old_start*, o bien se devuelve el valor NULL en caso de error. En este último caso, la variable errno puede tomar los valores siguientes: (1) EAGAIN (La región de memoria está bloqueada y no puede desplazarse); (2) EFAULT (La región de memoria especificada por old_start y old_len no formo parte del espacio de direccionamiento del proceso que llama); (3) EINVAL (old_start, old_len o new_len contiene un valor no válido (por ejemplo una dirección que no está alineada en una frontera de página)); (4) ENOMEM (El tamaño de la región de memoria no puede aumentarse, y MREMAP_MAYMOVE no se ha especificado).

Por lo que respecta a la función de biblioteca mprotect modifica los atributos de protección para una zona de memoria en el segmento de usuario utilizando las macros (PROT_NONE, PROT_READ, PROT_WRITE y PROT_EXEC) expuestas anteriormente. La implementación de esta función se basa en la llamada al sistema *mprotect*, que comprueba si una zona de memoria ha sido proyectada en este momento y si los nuevos atributos de protección son legales para la zona.

El programa ejemplo *EjemploMmap.cpp* utiliza la función *mmap* para acceder al contenido de un archivo. Abre un archivo, utiliza *mmap* para proyectar su contenido en memoria, y efectúa un bucle de visualización de cada byte contenido en el archivo.

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>

int main (int argc, char *argv[])
{
        int fd, i;
        struct stat st;
        char *addr;

        /* Control de argumentos */
        if (argc ! = 2)
        {
                fprintf(stderr, “Uso: %s nombre_de_archivo\n”, argv[0]);
                exit(1);
        }

        /* Obtención del tamaño del archivo */
        if (stat(argv[1], &st) == -1)
        {
                perror(“stat”);
                exit(2);
        }

        /* Apertura del archivo */
        fd = open(argv[l], O_RDONLY);
        if (fd == -1)
        {
                perror(“open”);
                exit(3);
        }
```

```
        /* Proyección del archivo en memoria */
        addr = (char*)mmap(NULL, st.st_size, PROT_READ, MAP_SHARED, fd, (off_t)0);
        if (addr == NULL)
        {
                perror("mmap");
                (void) close(fd);
                exit(4);
        }

        /* Cierre del archivo */
        close(fd);

        /* Bucle de visualización del contenido del archivo */
        for (i = 0; (i < st.st_size); i++)
                putchar(addr[i]);

        /* Liberación del archivo (de la proyección en memoria del archivo) */
        if (munmap(addr, st.st_size) == -1)
        {
                perror("munmap");
                (void) close(fd);
                exit(5);
        }

        exit(0);
}
```

Otro ejemplo que vamos a estudiar para comprobar el uso de las funciones que tiene Linux para proyectar el contenido de archivos en memoria (mmap() y munmap()) es el que vuelca un archivo especificado (como argumento) en format hexadecimal y ASCII. Al programa lo vamos a denominar *Dump.cpp*, y su código fuente podría ser el siguiente.

```
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/mman.h>

int main(int argc, char *argv[])
{
        int i, j, fd, fileSize;
        char *addr;
        unsigned char *cp, ch;

        /* Control de argumentos */
        if (argc != 2)
        { printf("Se debe especificar un archivo de entrada\n"); exit(1); }

        /* Proyecta el archivo de entrada especificado al espacio de direcciones del usuario */
        fd = open(argv[1], O_RDONLY);
        if (fd == -1)
        {
                perror("open");
                exit(1);
        }
```

```
        fileSize = lseek(fd, 0, SEEK_END);
        addr = (char*)mmap(NULL, fileSize, PROT_READ, MAP_PRIVATE, fd, (off_t)0);
        if (addr == NULL)
        {
                perror("mmap");
                exit(1);
        }
        close(fd);

        /* Visualiza el título en la salida por pantalla */
        printf("\nVolcado del archivo: %s (%d bytes)\n", argv[1], fileSize);

        /* Bucle principal para visualizar el volcado en hexadecimal y ASCII */
        cp = (unsigned char*)addr;
        for (int i = 0; (i < fileSize); i += 16)
        {
                printf("\n%08X: ", i);
                for (j = 0; j < 16; j++)
                {
                        if (i+j < fileSize )
                                printf("%02X ", cp[i+j]);
                        else
                                printf("   ");
                }
                printf(" ");
                for (j = 0; (j < 16); j++)
                {
                        ch = (i+j < fileSize) ? cp[i+j] : ' ';
                        if ((ch < 0x20) || (ch > 0x7E))
                                ch = '.';
                        printf("%c", ch);
                }
        }
        printf("\n\n");

        /* Liberación del archivo de entrada proyectado en memoria */
        if (munmap(addr, fileSize) == -1)
        {
                perror("munmap");
                close(fd);
                exit(5);
        }

        exit(0);
}
```

PRACTICA 2. LLAMADAS AL SISTEMA EN Linux.

2.1. Introducción y ejemplo básico de llamada al sistema
2.2. Objetivos de la práctica
2.3. Funcionalidad de la llamada al sistema (sys_generación)
2.4. Implementación a nivel de kernel del sistema operativo
2.5. Comprobación del funcionamiento

2.1. INTRODUCCIÓN Y EJEMPLO BÁSICO DE LLAMADA AL SISTEMA

Las funciones correspondientes a llamadas al sistema siempre empiezan por sys_, y se puede añadir en el fichero kernel/sys.c, donde está el código de otras llamadas al sistema, al final.

Sabemos que un proceso se ejecuta normalmente en modo usuario y debe pasar a modo *kernel* para ejecutar las *llamadas al sistema*. Una *llamada al sistema* se caracteriza por un nombre y un número único que la identifica. Este número puede encontrarse en el archivo <*usr/src/linux-source-2.6.28/arch/x86/include/asm/unistd_32.h*>. Por ejemplo, la llamada al sistema *open* tiene el número 5. El nombre se define en el archivo ensamblador *usr/src/linux-source-2.6.28/arch/x86/kernel/entry_32.S* que tiene al final un include a *syscall_table_32.S* (aunque este archivo se encuentra en un directorio dedicado a la arquitectura i386, existe para las otras arquitecturas), y se llama *sys_open*. La librería C proporciona funciones que permiten ejecutar una llamada al sistema. Estas funciones afectan al nombre de las llamadas al sistema. Basta, pues, para ejecutar una llamada al sistema, con llamar a la función correspondiente. En los programas fuente del *kernel*, una llamada al sistema posee el prototipo siguiente:

```
asmlinkage int sys_nombrellamada(lista de argumentso)
{
        /* código de la llamada al sistema */
}
```

El número de argumentos debe estar entre 0 y 5. La palabra clave *asmlinkage* es una macroinstrucción definida en el archivo <*linux/linkage.h*>:

```
#ifdef __cplusplus
#define asmlinkage extern "C"
#else
#define asmlinkage
#endif
```

Cuando un proceso ejecuta una llamada al sistema, llama a la función correspondiente de la librería C. Esta función trata los parámetros y hace pasar al proceso al modo *kernel*. En la arquitectura i386, este paso a modo *kernel* se efectúa de la siguiente forma:

- Los parámetros de la llamada al sistema se colocan en ciertos registros del procesador.
- Se provoca un bloqueo desencadenando la interrupción lógica 0x80.
- Este bloqueo proporciona el paso del proceso al modo *kernel*, el proceso ejecuta la función *system_call* definida en el archivo fuente *usr/src/linux-source-2.6.28/arch/x86/kernel/entry_32.S*, que tiene al final un include a *usr/src/linux-source-2.6.28/arch/x86/kernel/syscall_table_32.S*
- Esta función utiliza el número de llamada al sistema (transmitido en el registro *eax*) como índice en la tabla *sys_call_table*, que contiene las direcciones de las llamadas al sistema, y llama a la función del *kernel* correspondiente a la llamada al sistema.
- Al retornar de esta función, *system_call* vuelve a quien la ha llamado, este retorno vuelve a pasar al proceso a modo usuario.

En general, para dar de alta la función en el kernel debemos:

- Añadir al archivo *usr/src/linux-source-2.6.28/arch/x86/include/asm/unistd_32.h* el identificador de la llamada.
- Añadir en *usr/src/linux-source-2.6.28/arch/x86/kernel/syscall_table_32.S* el puntero a la función sys_nombrellamada que implementa la llamada al sistema. Puede ser necesario refrescar los

conocimientos de teoría sobre lo que es la *sys_call_table* y cómo se usa en el kernel, en la rutina de servicio de la interrupción software 0x80, en el mismo archivo.

- Actualizar el número de llamadas al sistema implementadas en *usr/src/linux-source-2.6.28/arch/x86/include/asm/unistd.h* al final de la lista de llamadas al sistema.

Para comprender bien el funcionamiento de una llamada al sistema, nada mejor que crear una. La llamada al sistema que vamos a crear como ejemplo consiste en tomar tres argumentos, hacer la multiplicación de los dos primeros y colocar el resultado en el tercero. Para empezar, es necesario declarar la llamada al sistema, y por tanto asignarle un número. Para declarar nuestra llamada *(sys_show_mult)*, hay que saber cuál es el número de llamadas al sistema del *kernel* con el que estamos trabajando, y después hay que añadir el primer número libre en el archivo *usr/src/linux-source-2.6.28/arch/x86/include/asm/unistd.h*:

Archivo **unistd_32.h**
/usr/src/linux-source-2.6.28/arch/x86/include/asm/unistd_32.h

```
#define __NR_show_mult          333 /* nueva llamada al sistema */
#define __NR_generacion         334 /* nueva llamada al sistema */
#define __NR_getgeneracion      335 /* nueva llamada al sistema */
```

Archivo **syscalls.h**
/usr/src/linux-source-2.6.28/arch/x86/include/asm/syscalls.h
Debajo de #ifdef CONFIG_X86_32

```
asmlinkage int sys_show_mult(int x, int y, int *res);
asmlinkage int sys_generacion(void);
asmlinkage int sys_getgeneracion(pid_t pid);
```

Archivo **syscall_table_32.c**
/usr/src/linux-source-2.6.28/arch/x86/kernel/syscall_table_32.S

```
.long sys_show_mult /* nueva llamada al sistema */
.long sys_generacion /* nueva llamada al sistema */
.long sys_getgeneracion /* nueva llamada al sistema */
```

Archivo **sys.c**
/usr/src/linux-source-2.6.28/kernel/sys.c
Se añaden las 3 funciones para las 3 llamadas descritas anteriormente:

```
asmlinkage int sys_show_mult(int x, int y, int *res)
{
  int noerror;
  int compute;
  /* Verificacion de la validez de la variable res */
  noerror = access_ok(VERIFY_WRITE, res, sizeof(*res));
  if (!noerror)
      return noerror;
  /* Calculo del resultado de la multiplicacion */
  compute = x*y;
 /* Copia el resultado en la memoria del usuario */
 put_user(compute, res);
 printk("Valor calculado: %d * %d = %d \n", x, y, compute);
 /* Llamada al sistema finalizada */
 return 0;
}
```

Una vez completados los pasos anteriores, sólo es necesario recompilar el kernel e instalarlo igual que en la práctica 1. Es muy importante revisar a conciencia el código de la llamada al sistema, puesto que recordemos que se ejecutará en modo kernel y si por algún motivo falla o se queda ejecutando un bucle infinito, el sistema no se recuperará. Recordemos que Linux no es preemptivo cuando se ejecuta en modo kernel. El sistema se detendrá y será necesario resetear en frío.

Compilacion del kernel
Al añadir una nueva llamada al sistema es necesario recompilar el kernel para que se incluyan, se realizan los mismos pasos que en la practica 1.

Una vez recompilado correctamente el kernel y reiniciada la máquina, es posible probar la nueva llamada al sistema: Para poder utilizar la llamada al sistema, es necesario declarar una función que ejecuta esta llamada al sistema. Esta función no existe en la librería C, y hay que declararla explícitamente. Varias macroinstrucciones que permiten definir este tipo de funciones se declarar en el archivos de cabecera <*syscalls.h*>. En el caso de una llamada al sistema que acepta tres parámetros, hay que utilizar la macroinstrucción *_syscall3*:

Comprobacion de las llamadas al sistema
Se crean 3 programas para probar las 3 llamadas al sistemas implementadas anteriormente.

La función show_mult es generada (expandida) por *_syscall3*, que inicializa los registros del procesador (el número de la llamada al sistema se coloca en el registro *eax* y los parámetros se colocan en los registros *ebx*, ecx y edx), a continuación desencadena la interrupción 0x80. De vuelta de esta interrupción, es decir, al volver de la llamada al sistema, el valor de retorno se comprueba. Si es positivo o nulo, se devuelve a quien la llama; en caso contrario, este valor contiene el código de error devuelto, entonces se guarda en la variable global errno y se devuelve el valor –1.

La función syscall() hace el mismo papel que anteriormente _syscallX. Con dicha función podemos llamar a una llamada al sistema especificando su índice de llamada en la tabla de llamadas al sistema y un conjunto de argumentos; y adicionalmente de deben de incluir #include <sys/syscall.h>.

Archivo **showmult.c**
Programa para la prueba de la llamada de sistema showmult:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/syscall.h>
#include <linux/kernel.h>
#define show_mult 333
int main(int argc, char** argv)
{
    int ret = 0;
    syscall(show_mult,2, 5, &ret);
    printf("Resultado: %d * %d = %d \n", 2, 5, ret);
    return 0;
}
```

2.2. OBJETIVOS DE LA PRÁCTICA

El objetivo principal de esta práctica es añadir una nueva funcionalidad al kernel de Linux mediante la implementación de una nueva llamada al sistema. Hay que abordar necesariamente esta implementación desde dos perspectivas:

- Añadir la llamada al sistema en el kernel del sistema operativo. Con esto conseguiremos que, cuando un proceso cargue cierto valor en el registro *eax* y lance a continuación la interrupción software 0x80, la rutina de servicio de esta interrupción ejecute la función del kernel que queramos.
- Crear la interfaz C de la llamada al sistema. Y con esto lograremos que cualquier programa pueda realizar la llamada al sistema mediante la invocación de una función normal de C con sus parámetros, etc.

…

2.3. FUNCIONALIDAD DE LA LLAMADA AL SISTEMA (sys_generacion)

La nueva llamada al sistema debe devolver la *generación que ocupa un proceso dentro del árbol genealógico de procesos*, es decir, el número de ancestros que hay que atravesar en el árbol de procesos hasta alcanzar la raíz: el proceso *init* (pid == 1).

Veamos un ejemplo analizando la salida del comando **ps l**:

...	PID	PPID	PRI	...	TIME	COMMAND
...	150	1	0	...	0:00	bash
...	556	150	0	...	0:00	xinit /root/.xinitrc
...	564	556	1	...	0:01	kwm
...	573	564	0	...	0:00	kaudioserver
...	579	564	0	...	0:04	kpanel
...	581	573	0	...	0:00	maudio −media 129

PID identifica al proceso, y PPID al padre de ese proceso. En este caso, si la llamada la invocara el proceso bash, con PID 150, el resultado debería ser 1, pues es hijo directo del proceso *init* (PID 1). La siguiente tabla muestra el resultado que se esperaría de la ejecución de la nueva llamada para los procesos anteriores (generados por la ejecución de la orden **ps l**):

PID	PPID	COMMAND	Resultado
556	150	xinit /root/.xinitrc	2
564	556	Kwm	3
573	564	kaudioserver	4
579	564	kpanel	4
581	573	maudio −media 129	5

Nombre de la llamada al sistema: **generacion** ⇒ obtiene la generación a la que pertenece el proceso invocante
Sinopsis de la llamada al sistema: int *generacion*(void);

Descripción de la llamada al sistema: **generacion** devuelve la profundidad dentro del árbol genealógico de procesos, es decir, el número de procesos que hay que atravesar desde el proceso *init* hasta alcanzar el proceso invocante.

2.4. IMPLEMENTACIÓN A NIVEL DE KERNEL DEL SISTEMA OPERATIVO

Una vez detallada la funcionalidad que se espera de la nueva llamada al sistema, vamos entrar en los detalles de implementación. Primero veremos cómo construir la función en C, que implementaremos dentro del kernel, y que será la que haga todo el trabajo, y después veremos cómo dar de alta dicha función dentro del kernel para ofrecerla como llamada al sistema.

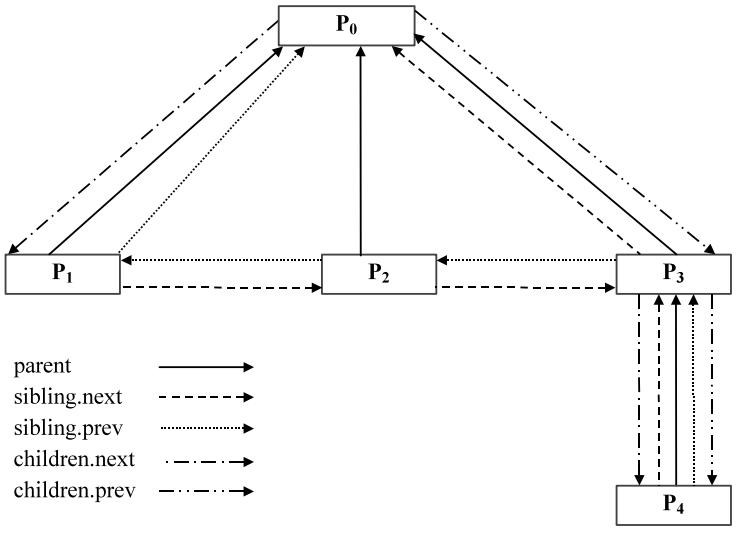
Todos los procesos del sistema tienen asociada, dentro del kernel, una estructura que contiene toda la información relevante de dicho proceso **struct task_struct** en el archivo cabecera include/linux/sched.h. Durante esta práctica denominaremos a dicha estructura *descriptor de proceso*. Se suele trabajar con punteros a la tabla de procesos, luego manejaremos a menudo variables de tipo puntero a descriptor de proceso (struct task_struct *). La macro *current* contiene el puntero correspondiente al proceso que está actualmente en ejecución.

El descriptor de un proceso **(struct task_struct)** contiene información sobre el proceso al que representa: su *pid*, el identificador de usuario, *uid*, el identificador de grupo, *gid*, los descriptores de archivos abiertos, y un largo etc. Entre toda la información que contiene cabe destacar los siguientes campos en …/include/linux/sched.h. Dicho descriptor tiene múltiples campos, con toda la información necesaria para la realización de la llamada al sistema. Y con estos campos nos permiten trazar el árbol genealógico de un proceso hasta el origen (init).

- pid: El PID del proceso
- real_parent: El proceso padre original.
- parent: El proceso padre.
- children: Lista de hijos.
- sibling: Lista de procesos hermanos.

```
pid_t pid;
/*
        * pointers to (original) parent process, youngest child, younger sibling,
        * older sibling, respectively. (p->father can be replaced with
        * p->parent->pid)
*/
struct task_struct *real_parent;  /* real parent process (when being debugged) */
struct task_struct *parent;       /* parent process */
/*
        * children/sibling forms the list of my children plus the
        * tasks I'm ptracing.
*/
struct list_head children;        /* list of my children */
struct list_head sibling;         /* linkage in my parent's children list */
```

El significado de los campos en el kernel de Linux 2.6 es el siguiente: *pid* (el PID del proceso); *real_parent* (puntero que apunta al descriptor del proceso que creó P o al descriptor del proceso 1 *(init)* si el proceso padre no existe. Por tanto, cuando un usuario lanza un proceso en background y cierra (exit) la shell, el proceso en background se convierte en hijo de *init*); *parent* (puntero que apunta al padre actual de P y normalmente coincide con el de *real_paren*t, salvo en determinados casos); *children* (el principio de la lista que contiene todos los hijos creados por P); *sibling* (los punteros a los elementos siguiente (next) y anterior (previous) en la lista de procesos *hermanos*, aquellos que tienen el mismo padre P. A los procesos que son todos hijos del mismo padre, se les denomina siblings (hermanos), es decir, cuando un proceso creó varios hijos (children), esos hijos están bajo la relación sibling). En la siguiente figura se pueden observar las relaciones (filiaciones) *parent* y *sibling* (en el kernel de Linux 2.6) de un grupo de procesos en el que un proceso P_0 ha creado sucesivamente tres procesos P_1, P_2 y P_3, y además el proceso P_3 ha creado al proceso P_4.



Para recorrer la lista de procesos se empleará la macro *for_each_process*, definida en …/include/linux/sched.h. En caso de que no exista un proceso, la función devolverá un error de tipo –ESRCH indicando que no existe dicho proceso (los códigos de error están definidos en …/include/asm-386/errno.h).

Estos campos permiten organizar los procesos en un árbol genealógico (relaciones padre-hijo-hermano) de un proceso hasta el origen *(init)*. Con el comando **pstree** podemos obtener un árbol similar que muestra únicamente la relación de parentesco. Como ejercicio, probar dicho comando y estudiar la estructura de procesos que devuelve, identificando cuál campo que nos servirá para recorrer el árbol genealógico de procesos (como avance, preste especial atención a *parent*).

Implementación de la función interna **sys_generacion**. La función que implementará la nueva llamada al sistema **sys_generacion** se añade en archivo …/kernel/sys.c, que contiene la implementación de algunas de las llamadas al sistema.

```
...
asmlinkage int sys_generacion(void)
{
        int generacion;
        ...
        /* return generacion; */
} /* end sys_generacion */
```

Una vez implementada la función, y dada de alta en el kernel del sistema, sólo queda recompilar el kernel de Linux, como se hizo en la primera práctica, y rearrancar el sistema.

2.5. COMPROBACIÓN DEL FUNCIONAMIENTO

Para comprobar el correcto funcionamiento de la nueva llamada al sistema, se realizará un programa de prueba que invoque dicha llamada a través de su interfaz en C. El programa deberá devolver a qué generación pertenece él mismo.

```
# mi_generacion
Soy de la generación 5.
```

Como ejercicio de ampliación se propone modificar la llamada al sistema anterior (**sys_generacion**) de forma que reciba un pid como parámetro y devuelva el número de generación del proceso con ese *pid* (**sys_getgeneracion**). Para recorrer la lista de procesos se puede utilizar la macro *for_each_process* que se encuentra declarada en el archivo …/sched.h. Es importante examinar, estudiar y entender la definición de esta macro.

Veamos un ejemplo de uso:

```
struct task_struct *p;
...
for_each_process(p)
{
        if (p−>pid == ALGO)
        {
                /* HACER LO QUE SEA NECESARIO */
                break;
        } /* endif */
} /* end for_each_process */
```

En caso de que no exista un proceso con el *pid* indicado, la función devolverá un error del tipo −ESRCH que está declarado en *.../errno.h*, indicando que no existe dicho proceso.

Al igual que se ha procedido anteriormente, se deberá construir una nueva orden **generacion**, que admita como parámetro el pid de un proceso, y que devuelva un mensaje indicando, o bien la generación de dicho proceso, o bien el mensaje de error correspondiente, indicando el valor de la variable errno.

Utilice **pstree −p** para ver qué procesos tenéis en marcha y probad la orden con varios de ellos. Probad también con números de proceso inexistentes y con el proceso *init*, para comprobar que se ha tenido en cuenta adecuadamente estos casos particulares.

PRACTICA 3. MÓDULOS CARGABLES DEL kernel.

3.1 INTRODUCCIÓN

Fundamentalmente, el kernel de Linux está constituido por múltiples componentes que no todos son necesarios para los usuarios en todo momento. Por ello, es por lo que al crearse el kernel, Linux solicita al usuario que especifique los elementos que desea incluir en el kernel. Evidentemente, el objetivo de esta operación es insertar únicamente los gestores necesarios en función de la configuración de la máquina y de su uso. De este modo, el tamaño del kernel es lo más reducido posible en función de la máquina. Cuanto menor es el tamaño del kernel, queda más memoria disponible para el usuario. Además, el arranque de una máquina con un kernel especialmente adaptado, es decir, únicamente con los gestores de dispositivos que posee la máquina, es mucho más rápido.

Sin embargo, cualquier modificación del kernel, como la inclusión o la supresión de un gestor de dispositivo, un sistema de archivos, implica la recompilación del kernel. Esto era cierto en las primeras versiones de Linux, hasta la implementación de los *módulos cargables*.

El objetivo de los *módulos cargables* es generar en un primer momento un kernel mínimo, y cargar los gestores de manera dinámica en función de las necesidades. Esto permite, en un momento dado, tener un kernel "extendido". Sin embargo, hay que destacar que el kernel debe estar configurado para que gestione los módulos (Loadable module support). Este sistema de módulos cargables existe también en otros sistemas operativos UNIX como por ejemplo Solaris, pero bajo una forma diferente.

Linux proporciona un primer método de uso de los módulos: se trata de la versión inicial de estos módulos. El principio es permitir al superusuario cargar y descargar manualmente los módulos según las necesidades. Esta técnica es bastante pesada de manipular porque toda operación debe efectuarse manualmente. Además, el usuario normal no tiene derechos suficientes para efectuar esta operación. Linux proporciona principalmente tres ordenes que permiten manipular los módulos: *insmod, lsmod* y *rmmod.*

La orden *insmod* permite efectuar la carga del módulo. *lsmod* se limita a mostrar el contenido del archivo */proc/modules*. *rmmod* descarga el módulo deseado.

La orden *lsmod* indica no sólo el nombre de cada uno de los módulos cargados en memoria, sino también el número de páginas de memoria ocupadas por el módulo, así como el número de procesos que utilizan este módulo. Una página de memoria ocupa 4 KB, por lo que resulta fácil calcular la memoria economizada cuando estos módulos no están cargados.

El sistema de carga dinámica permite automatizar las cargas de los distintos módulos en función de la demanda. Su implementación precisa la activación en la compilación de la opción CONFIG_KERNELD y los IPC System V. El demonio *kerneld* utiliza las colas de mensajes para comunicarse con el kernel: la carga o descarga de un módulo la realiza *kerneld,* pero las órdenes se envían desde el kernel mediante una cola de mensajes especial. La implementación de esta técnica en una máquina implica el lanzamiento del programa *kerneld* al arrancar el sistema. También es necesario efectuar ciertas operaciones para construir la lista de módulos cargables instalados. Los módulos cargables dinámicamente necesitan dos programas al inicializarse la máquina: (1) *depmod,* que permite generar un archivo de dependencias basado en los símbolos encontrados en el conjunto de los módulos. Estas dependencias se usarán posteriormente en la segunda orden; (2) *modprohe,* que permite cargar un módulo o un grupo de módulos pero también cargar los módulos básicos necesarios para el correcto inicio de la máquina (como NFS, etc.).

Para manejar los disquetes, estos pueden montarse con la orden *mount(8)* o bien usar las "mtools": *mdir(1)*, *mformat(1)*, *mcopy(1)*, *mdel(1)*, etc. Podemos guardar los archivos en discos flexibles, ya que sólo es necesario conservar de una sesión a otra los archivos fuente que se hayan modificado, no el resto de archivos del kernel, ni ejecutables, ni archivos objeto (.o). Los archivos modificados siempre serán de un tamaño perfectamente manejable. Para manejar los disquetes pueden montarse con la orden *mount(8)* o bien usar las "mtools": *mdir(1)*, *mformat(1)*, *mcopy(1)*, *mdel(1)*, etc. Recuerde siempre borrar todo y usar *shutdown −r now* o pulsar Ctrl−Alt−Supr antes de apagar (tras salir del entorno gráfico).

3.2. LOS MÓDULOS CARGABLES EN Linux (Loadable Kernel Modules, LKM)

El kernel de Linux está organizado siguiendo una arquitectura monolítica, en la cual, todas las partes del kernel del sistema operativo (sistemas de archivos, manejadores de dispositivos, protocolos de red, etc.) están enlazadas como una sola imagen (normalmente el archivo /vmlinuz) que es la que se carga y ejecuta en el arranque del sistema.

Esta estructura podría dar lugar a un sistema poco flexible, ya que cualquier funcionalidad que se le quisiera añadir al kernel del sistema requeriría una recompilación completa del mismo. Aún así, la filosofía de *open source* (fuentes abiertos) hace Linux mucho más flexible que otros sistemas operativos en la que los fuentes no están disponibles. No obstante, la recompilación total del kernel puede resultar engorrosa en las fases de desarrollo de nuevos manejadores de dispositivo, ampliaciones no oficiales del kernel, etc.

Esta limitación desapareció con la incorporación, en la versión 2.0 de Linux, del soporte para la carga dinámica de módulos en el kernel. Esta nueva característica permite la "incorporación en caliente" de nuevo código al kernel del sistema operativo, sin necesidad de reinicializar el sistema.

La práctica aborda el desarrollo de pequeños módulos. Los módulos son "trozos de sistema operativo", en forma de archivos objeto (.ko), que se pueden insertar y extraer en tiempo de ejecución (forma dinámica). Dichos archivos .ko se pueden obtener directamente como resultado de la compilación de un archivo .c *(gcc −c prog.c)*, o como la unión de varios archivos .ko enlazados *(ld −r f1.o f2.o)*. La única característica especial que deben tener estos archivos, es la de incorporar las funciones *init_module* y *cleanup_module*. Más adelante veremos su utilidad.

Una vez desarrollado un módulo e insertado en el kernel, su código pasa a ser parte del propio kernel, y por lo tanto se ejecuta en el modo supervisor del procesador (nivel de privilegio 0 en la arquitectura i386), con acceso a todas las funciones del kernel, a las funciones exportadas por módulos previamente insertados, y a todo el hardware de la máquina sin restricciones.

La única diferencia con código enlazado en el kernel es la posibilidad de extraer el módulo una vez ha realizado su labor o ha dejado de ser útil, liberando así todos los recursos utilizados. Naturalmente, para insertar un módulo en el kernel se debe hacer en modo superusuario (supervisor), puesto que si un usuario normal pudiera insertar módulos, esto se convertiría en un serio y evidente problema de seguridad.

Dado que el código de un módulo puede utilizar cualquier función del kernel, pero no ha sido enlazado en tiempo de compilación con él, las referencias a las funciones del kernel no están resueltas. Por tanto, cuando se inserta un módulo en el kernel se deben seguir los siguientes pasos:

- Obtener las referencias a funciones ofrecidas por el módulo.
- Incorporar dichas referencias al kernel, como referencias temporales, que desaparecerán con la extracción del módulo.
- Resolver las referencias a las funciones no resueltas en el módulo, ya sean llamadas a funciones del kernel, como a llamadas a funciones de otros módulos.
- Insertar el módulo en la zona de memoria correspondiente al kernel.
- Finalmente, invocar a la función *xxx_init* registrada a través de la macro *module_init()* como punto de entrada del nuevo módulo.

La extracción de un módulo del kernel se realiza mediante una secuencia similar a la anterior, pero en orden inverso, donde antes de extraer el módulo se invoca a la función *xxx_exit*, registrada a través de la macro *module_exit()* como punto de salida del módulo.

3.3. PROGRAMACIÓN DE MÓDULOS CARGABLES

La función *xxx_init* nos van a permitir inicializar el módulo al insertarlo en el kernel (equivaldría a la función *main* de un programa en C). Por otro lado, *xxx_exit* se usará para liberar los recursos utilizados cuando se vaya a extraer.

A continuación, vamos a estudiar la forma de generar el código correspondiente al módulo. Naturalmente, el kernel deberá tener compilado el soporte para módulos, o no podremos insertarlos. Un módulo lo generaremos a partir de un archivo fuente C, del estilo del siguiente.

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

int n = 1;
module_param (n, int, 0644);

static int ejemplo_init(void)
{
        printk("Entrando. n = %d\n", n);
        return 0;
}

static void ejemplo_exit(void)
{
        printk("Saliendo.\n");
}

module_init(ejemplo_init);
module_exit(ejemplo_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Sr. X");
```

Este programa podría estar almacenado en el archivo ejemplo.c. Es necesario además un archivo Makefile con la siguiente línea, que indica el nombre del módulo a generar (observe que tiene extensión .o y no .ko):

```
obj-m := modulo.o
```

A continuación compilaríamos el programa empleando el programa make de una manera particular para la creación de módulos, con una sentencia como la siguiente

```
make -C /usr/src/… SUBDIRS=$PWD modules
```

Es necesario indicar que el directorio kernel-source-2.6/… en el ejemplo anterior deberá adecuarse al lugar exacto donde tengamos descomprimido el código fuente del kernel, que además deberá estar configurado. Esta sentencia genera el archivo ejemplo.ko, que ya es un módulo insertable.

En el programa anterior se emplea la función printk(). Esta función es similar a printf(), salvo que en lugar de imprimir en la salida estándar imprime en un buffer de mensajes del kernel o kernel ring buffer. En este buffer escribe normalmente el kernel todo tipo de errores, notificaciones o eventos que ocurren a nivel del kernel. Se puede ver su contenido ejecutando el comando dmesg, consultando /proc/kmsg o visualizando el fichero /var/log/messages. Por ejemplo podemos emplear la orden tail –f /var/log/messages para ver cómo

progresa dicho buffer de forma constante. El manejo de los mensajes del kernel brinda otras posibilidades, algunas de ellas muy complejas. Una propiedad de interés es que se puede indicar la prioridad o la importancia del mensaje mediante una indicación con tres caracteres al principio de la cadena de la forma <n>, donde n es un número de 0 a 7 que indica el tipo de mensaje. Normalmente el núcleo se configura para que sólo se muestren por consola los mensajes de prioridad superior a 6.

KERN_EMERG	System is unusable
KERN_ALERT	Action must be taken immediately
KERN_CRIT	Critical conditions
KERN_ERR	Error conditions
KERN_WARNING	Warning conditions
KERN_NOTICE	Normal but significant condition
KERN_INFO	Informational
KERN_DEBUG	Debug−level messages

Normalmente, el kernel está configurado para mostrar por la consola activa los mensajes de prioridad superior a 6. (Los terminales gráficos no son consolas, a no ser que se hayan lanzado explícitamente como tales).

3.4. UTILIZACIÓN DE LOS MÓDULOS

Como ya hemos comentado anteriormente, para insertar o extraer módulos del kernel debemos tener permisos de superusuario o supervisor. La inserción (carga) de un módulo se lleva a cabo mediante la orden *insmod*, que realizará todas las acciones comentadas antes para insertar el código en el kernel. Ejecute, desde una consola, la siguiente orden y observe qué ocurre en el buffer de salida del kernel tras la inserción:
insmod ejemplo.ko

Observe qué ocurre en el buffer de salida del kernel tras la inserción. Acabamos de instalar ejemplo y ejecutar su función *init_module()*. Si se le pasa a *insmod* un nombre de archivo sin ruta ni extensión, se busca en los directorios estándar (ver *insmod(8)*).

Para ver los módulos que están cargados y cierta información sobre ellos, emplearemos la orden *lsmod*. Y finalmente, para extraer un módulo del kernel, emplearemos *rmmod*.

rmmod ejemplo

Para pasarle parámetros a un módulo, hay que asignar valores a las variables globales que hemos declarado como parámetros en la macro *module_param*. Esta macro recibe como primer argumento la variable, como segundo argumento el tipo de dicha variable y finalmente como tercer argumento los permisos del archivo correspondiente en *sysfs*, aspecto que de momento obviaremos.

Por ejemplo, en el caso que nos ocupa podríamos cargar el módulo de la siguiente forma.

insmod ejemplo.ko n = 4

Empleando la orden *modinfo ejemplo.ko* podemos averiguar los datos necesarios sobre los parámetros del módulo.

3.5. ENUNCIADO DE LA PRÁCTICA

El objetivo de la práctica consiste en la **implementación** de dos módulos cargables en el kernel de Linux, para observar ciertas particularidades de las dependencias entre módulos y su interacción. Estos módulos, que denominaremos *acumulador* y *cliente*, deberán atender al siguiente comportamiento:

- Ambos módulos (*acumulador* y *cliente*) han de mostrar cuando los insertemos (carga) o extraigamos (descarga), un mensaje informativo indicando el instante de la inserción y de la extracción (en número

de segundos desde el uno de enero de 1970). Podemos obtener el instante actual consultando la variable *xtime* del kernel, declarada en kernel/sched.h. El tipo de esta variable es *struct timeval* y está definido en include/linux/time.h (por tanto, debe incluir este archivo en los módulos).

- El módulo *acumulador* tendrá definida una función *void acumular(int i)*, que vaya sumando a una variable global del módulo el valor que se le pase como parámetro. Asimismo deberá ofrecer una función *int llevamos(void)* que permita consultar el valor de dicha variable global. La variable empieza valiendo cero, y cuando se extraiga el módulo, se mostrará el valor acumulado. Es importante que en el módulo acumulador se exporten las funciones para que puedan ser utilizadas por otros módulos, ya que por defecto todas las funciones que definamos serán privadas del módulo. Para ello, se empleará la macro *EXPORT_SYMBOL(simbolo)*.
- El módulo *cliente*, al ser insertado, llamará a la función *acumular()* del módulo acumulador, pasándole un valor igual al que le pasemos al módulo cliente al insertarlo. Cuando se extraiga este módulo, mostrará cuánto llevamos acumulado hasta el momento, haciendo uso de la función *llevamos()*. El módulo *cliente*, al ser insertado (cargado), debe llamar a la función *acumular()* del módulo *acumulador*, pasándole un valor a acumular igual al parámetro que le pasemos al módulo *cliente* al insertarlo (cargarlo). El módulo *cliente*, al ser extraído, debe llamar a la función *llevamos()* del módulo *acumulador* e imprimir el resultado acumulado en su mensaje de salida.

Como tenemos dos módulos será necesario un Makefile que permita construir ambos módulos. En este caso se especificará de la siguiente manera:

```
obj-m := modulo1.o
obj-m += modulo2.o
```

3.6. COMPROBACIÓN DEL FUNCIONAMIENTO

Compruebe que los módulos implementados se compilan correctamente. A continuación, inserte el módulo *acumulador*, y el módulo cliente varias veces, observando que el comportamiento es el correcto. Finalmente, extraiga el módulo *acumulador*, compruebe el mensaje y observe que el resultado final también es el correcto.

Después, observe qué sucede si:

- Intentamos insertar el *cliente* sin que esté el *acumulador* insertado.
- Intentamos extraer el acumulador estando el cliente insertado.

Explicando lo ocurrido en ambos casos.

Usa *lsmod* cada vez que inserte y extraiga los módulos, para asegurarse de que todo funciona correctamente. Observe también el estado del sistema cuando los módulos están insertados, empleando la misma orden.

PRACTICA 4. AÑADIR FUNCIONALIDAD AL KERNEL DE Linux UTILIZANDO MÓDULOS CARGABLES.

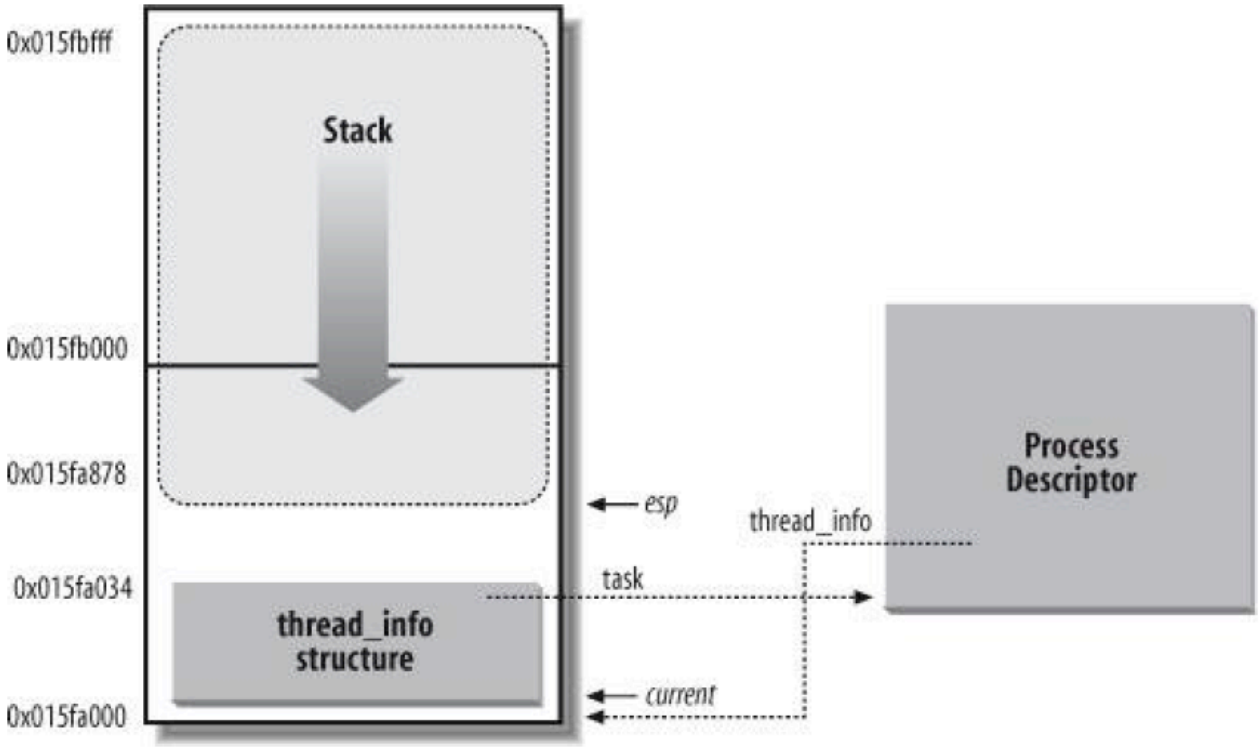
Con el objetivo de añadir más funcionalidades al kernel de Linux que se dispone en el laboratorio de prácticas (Laboratorio de Análisis y Desarrollo de Software), se plantea la implementación de los siguientes módulos cargables (para que interactúen con procesos, muestre el espacio de direcciones virtual de Linux, implemente un driver de dispositivo modo carácter, y oculte archivos en el sistema de archivos Ext2, interceptando llamadas al sistema).

4.1. VISUALIZACIÓN DE LOS DESCRIPTORES DE PROCESOS

Sabemos que los procesos son entidades dinámicas cuyo tiempo de vida fluctúa entre unos pocos milisegundos hasta meses. Por tanto, el kernel debe poder manejar muchos procesos al mismo tiempo, y los descriptores de procesos se almacenan en memoria dinámica más bien que en la zona de memoria permanentemente asignada al kernel. Linux almacena dos estructuras de datos para cada proceso en una única zona de memoria de 8 KBytes: la información del thread (*thread_info*) y la pila (*stack*) del proceso en modo kernel.

Además, también sabemos que un proceso en modo kernel accede a una pila contenida en el segmento de datos del kernel, que es diferente de la pila utilizada por el proceso en modo usuario. Debido a que los caminos de control del kernel hace poco uso de la pila, únicamente unos pocos miles de bytes de la pila del kernel son requeridos. Por lo que, 8 KBytes es un espacio más que suficiente para la pila y el descriptor de proceso.

En la siguiente figura podemos observar cómo las dos estructuras, están almacenadas en una zona de memoria de 2 páginas (2-page) de 8 KBytes. El *struct thread_info* reside al principio de la zona de memoria, mientras que la pila está localizada al final de la zona de memoria crece hacia abajo desde el final. Debemos destacar un matiz importante, que consiste en la posibilidad (y probabilidad) de que la pila pueda expandirse hacia abajo lo suficiente para sobrescribir alguna parte del struct thread_info, con el riesgo de dejar colgado el sistema operativo (estado corrupto). Además, también podemos observar que struct thread_info y struct task_struct están mutuamente enlazadas por medio del campo *task*.



El registro *esp* es el puntero de la pila de la CPU, que se utiliza para direccionar la parte más alta de la pila. En procesadores Intel, la pila empieza al final y crece hacia el principio de la zona de memoria. Justo después de conmutar de modo usuario a modo kernel, la pila del kernel de un proceso está siempre vacía, y por tanto el registro EPS apunta al byte inmediatamente a continuación de la zona de memoria del stack.

El valor del registro *esp* se decrementa tan pronto como se escribe en la pila. Ya que el thread_info es 52 bytes de longitud, la pila (stack) del kernel puede entonces expandirse hasta 8140 bytes. El lenguaje C (a través de una estructura de datos *union*) permite representar convenientemente el thread_info y la pila del kernel de un proceso por medio de la siguiente unión:

```
include/linux/sched.h
union thread_union {
        struct thread_info thread_info;
        unsigned long stack[2048]
};
```

Además, la estructura thread_info está definidad en x86 en el archivo cabecera */asm/thread_info.h* como

```
struct thread_info {
        struct task_struct *task;
        struct exec_domain *exec_domain;
        unsigned long flags;
        unsigned long status;
        __u32 cpu;
        __s32 preempt_count;
        mm_segment_t addr_limit;
        struct restart_block restart_block;
        unsigned long previous_esp;
        __u8 supervisor_stack[0];
};
```

En la figura anterior, el descriptor del proceso se almacena empezando en la dirección 0x015fa000 y la pila se almacena empezando en la dirección 0x015fc000. El valor del registro *esp* apunta a la parte superior de la

pila en 0x015fa878. El kernel utiliza macros para asignar (alloc_thread_info) y liberar (free_thread_info) las zonas de memoria de 8 KBytes que almacenan la estructura thread_info y una pila del kernel (stack); estando declaradas ambas macros en el archivo cabecera include/asm-i386/processor.h

Debida a la asociación creada entre el descriptor del proceso y la pila en modo kernel nos proporciona un beneficio importante en términos de eficiencia: *el kernel puede obtener fácilmente el puntero al descriptor del proceso, del proceso que actualmente se está ejecutando en la CPU a través del valor del registro esp*. De hecho, ya que la zona de memoria es de 8 KBytes (2^{13} bytes) de tamaño, todo lo que el kernel tiene que hace es enmascarar los 13 bits menos significativos del valor del registro *esp* para obtener la dirección base del descriptor del proceso. Esto se realiza por medio de la macro *current* (include/asm-i386/current.h), que produce instrucciones en lenguaje ensamblador como las que se muestran a continuación:

```
movl $0xffffe000, %ecx /* or 0xfffff000 for 4KB stacks */
andl %esp, %ecx
movl %ecx, p
```

Después de ejecutar estas tres instrucciones, *p* contiene el puntero al descriptor del proceso del proceso que actualmente se está ejecutando en la CPU y que ejecuta dicha instrucción. La macro *current* a menudo aparece en el código del kenrel como un prefijo a los campos del descriptor del proceso, como por ejemplo, current->pid devuelve el ID del proceso que actualmente se está ejecutando en la CPU

El **EJERCICIO** en sí consiste en implementar un módulo que *visualice* el objeto (unión) de Linux "thread_union". Este módulo genera un pseudo-archivo denominado "visual" en el directorio /proc, que permita a los usuarios visualizar una imagen de objeto de Linux "thread_union", mostrando tamaños y localización relativa de la información del thread (thread_info) y las zonas de pila (stack) como elementos de dicha unión, y además una indicación de la zona de memoria "basura" (garbage area) que separa estas dos estructuras de datos que forman la unión.

4.2. VISUALIZACIÓN DEL ESPACIO DE DIRECCIONES VIRTUAL BAJO Linux

Con este módulo se pretende implementar una visualización de los 4 GBytes de espacio de direcciones virtual bajo Linux, basado en entradas de directorio de tablas de página (page-directory entries). Además, tanto el directorio de tablas de pagina como las tablas de páginas pueden incluir hasta 1024 entradas.

La granularidad de los datos es de 4 MBytes (que es el tamaño del marco de página soportado por la MMU en la arquitectura i386 a partir del modelo Pentium (Paginación Extendida, *extended paging*, que permite marcos de página de 4 MBytes en lugar de marcos de página de 4 KBytes cuando el directorio intermedio de tablas de página se omite)). Es decir, que los procesadores de la arquitectura i386 pueden mapear marcos de página de 4 KBytes (1024 entradas en el directorio de tablas de páginas, por lo que puede haber hasta 1024 tablas de páginas y las tablas de página pueden apuntar hasta 1024 marcos de página de 4 Kbytes cada uno; con lo que tendríamos un total de 2^{20} marcos de páginas de 4 KBytes cada uno = 4 GBytes) y de 4 MBytes (el espacio de direcciones completo de 4 GBytes está subdividido en 1024 páginas de 4 MBytes), aunque nosotros en la práctica sólo utilizaremos estos últimos.

Los marcos de página que no se puede visualizar se mostraría con el carácter "-", mientras que marcos de página que si son proyectados (visualizados) se mostrarían con un carácter ("3" <marco de página del superusuario>·o "7" <marco de página del usuario>) basado en bits de la entrada en el directorio de tablas de páginas. Para utilizar este módulo, debemos compilar e instalar el archivo objeto del módulo (que denominaremos dirpag.c) y entonces ejecutar el siguiente comando desde la shell del sistema: # cat /proc/dirpag

Para implementar el módulo, debemos recordar que hay cuatro tipos de direcciones:

- **Direcciones lógicas**. Generadas por el proceso, cada dirección lógica consiste en un selector de segmento y un desplazamiento (offset) que denota la distancia del principio del segmento a la dirección actual.
- **Direcciones lineales (direcciones virtuales)**. Obtenidas tras aplicar una transformación a la dirección lógica por parte de la MMU. 32 bits se pueden utilizar para direccionar 4 GBytes (es decir 4294967296 direcciones físicas de memoria). Las direcciones lineales se representan normalmente en hexadecimal, su rango de valores va desde 0x00000000 hasta 0xffffffff. Las direcciones lineales y las lógicas son utilizadas internamente por la CPU. La CPU intenta una conversión a dirección física durante el acceso a una dirección virtual.
- **Direcciones físicas**. Referencian la memoria física. Se obtienen tras aplicar una transformación por parte de la MMU. Estas direcciones existen en el bus de memoria así como en la CPU. Éstas son las direcciones físicas utilizadas por la CPU para manejar físicamente el bus de datos.
- **Direcciones de bus**. Estas direcciones corresponden a las direcciones de memoria (direcciones físicas de los buses PCI, es decir, son aquellas que el dispositivo PCI tiene que generar para poder direccionar una posición de memoria) utilizadas por todos los dispositivos hardware excepto la CPU para manejar el bus de datos.

Las transformaciones y el formato de las direcciones dependen de la arquitectura. En Linux los espacios de direcciones lógico y lineal son idénticos. En la arquitectura i386 (PC), las direcciones de bus coinciden con las direcciones físicas (en otras arquitecturas, estas dos clases de direcciones son diferentes, por ejemplo en las SPARC de Sun). Además, las direcciones virtuales se utilizan para poder acceder a zonas de memoria desde el kernel, las direcciones de bus se utilizan para acceder a memoria desde un dispositivo PCI; y las direcciones físicas no se utilizan directamente, sólo como parámetro para funciones de manejo de memoria.

En Linux el espacio de direcciones lineales se divide en dos partes:

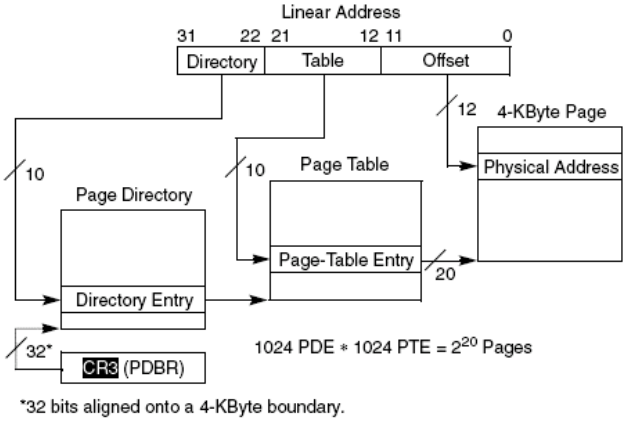
- Direcciones lineales desde 0x0000000 a PAGE_OFFSET – l, accesibles en cualquier modo (usuario o kernel).
- Direcciones lineales desde PAGE_OFFSET hasta 0xffffffff, sólo en modo kernel.

Generalmente PAGE_OFFSET se inicializa a 0xc0000000 (3Gb). Las primeras 768 entradas del directorio de tablas de páginas (que mapean los primeros 3Gb) dependen de cada proceso. El resto de entradas son las mismas para todos los procesos. El mapeado final proyecta las direcciones lineales empezando en PAGE_OFFSET en direcciones físicas empezando en 0.

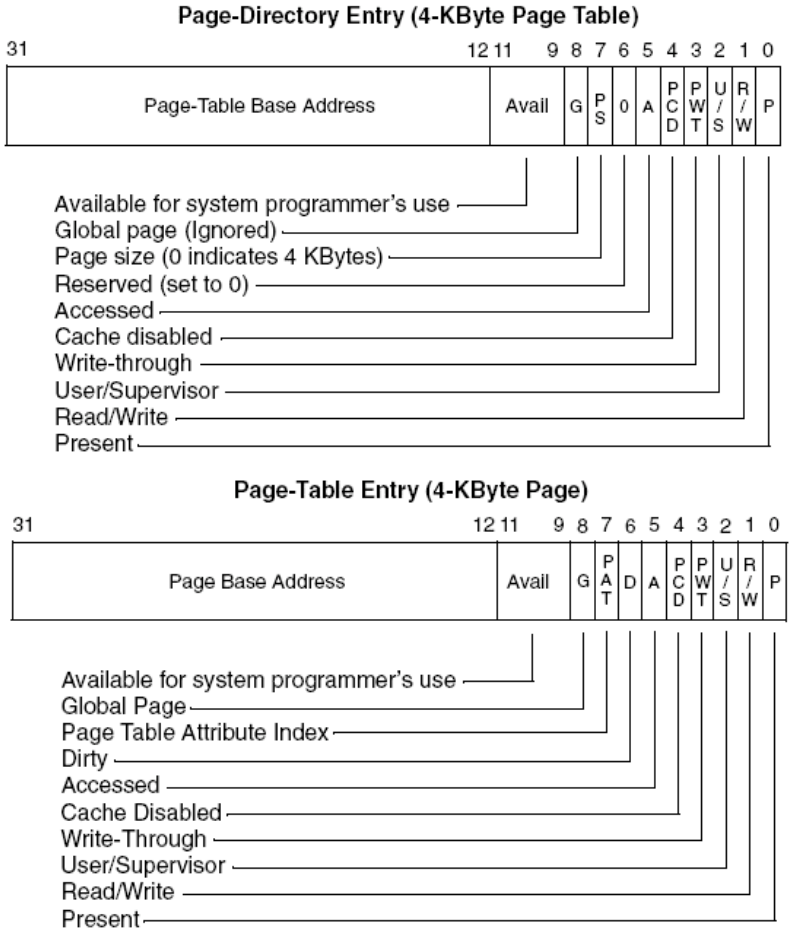
La dirección del directorio de tablas de páginas se almacena en el registro **CR3** del procesador (almacena la dirección física que contiene el primer nivel de la tabla de páginas (directorio de tablas de páginas)).

La función *phys_to_virt()* convierte una dirección física en dirección virtual y tiene el siguiente prototipo:
#include <asm/io.h>
unsigned long *phys_to_virt*(volatile void *address);

En la siguiente figure podemos observar como se transforma direcciones lineales en direcciones físicas (páginas de 4 KBytes) para la gestión de memoria en la arquitectura i386.



Por último, podemos observar en la siguiente figura el formato que tienen las entradas en el directorio de página y la tabla de páginas para páginas de 4 Kbytes y direcciones físicas de 32 bits.



4.3. VISUALIZACIÓN DEL MAPA DE MEMORIA VIRTUAL DE UN PROCESO EN Linux

Sabemos que un proceso en Linux tiene asociado un descriptor de proceso (struct task_struct), y que uno de los campos es un puntero al descriptor de memoria (struct mm_struct). El primero definido en /linux/sched.h y el segundo en /linux/mm.h. Ahora nos centraremos en la estructura del segundo.

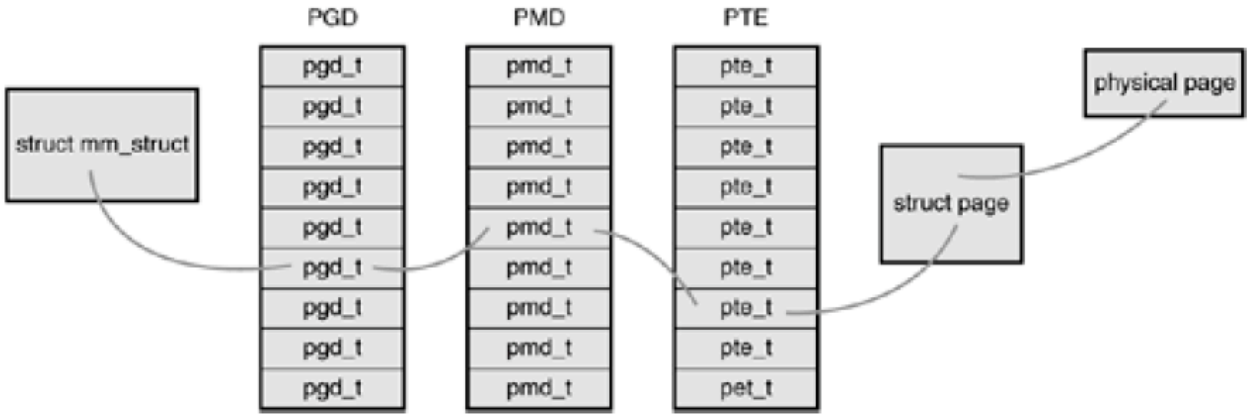
```
struct task_struct {
   pid_t pid;
   char comm[16];
   struct mm_struct *mm;
   /* plus many additional fields */
};

struct mm_struct {
   struct vm_area_struct *mmap; /* list of memory areas */
   struct rb_root mm_rb; /* red-black tree of VMAs */
   struct vm_area_struct *mmap_cache; /* last used memory area */
   unsigned long free_area_cache; /* 1st address space hole */
   pgd_t *pgd; /* page global directory */
   atomic_t mm_users; /* address space users */
   atomic_t mm_count; /* primary usage counter */
   int map_count; /* number of memory areas */
   struct rw_semaphore mmap_sem; /* memory area semaphore */
   spinlock_t page_table_lock; /* page table lock */
   struct list_head mmlist; /* list of all mm_structs */
   unsigned long start_code; /* start address of code */
   unsigned long end_code; /* final address of code */
   unsigned long start_data; /* start address of data */
   unsigned long end_data; /* final address of data */
   unsigned long start_brk; /* start address of heap */
   unsigned long brk; /* final address of heap */
   unsigned long start_stack; /* start address of stack */
   unsigned long arg_start; /* start of arguments */
   unsigned long arg_end; /* end of arguments */
   unsigned long env_start; /* start of environment */
   unsigned long env_end; /* end of environment */
   unsigned long rss; /* pages allocated */
   unsigned long total_vm; /* total number of pages */
   unsigned long locked_vm; /* number of locked pages */
   unsigned long def_flags; /* default access flags */
   unsigned long cpu_vm_mask; /* lazy TLB switch mask */
   unsigned long swap_address; /* last scanned address */
   unsigned dumpable:1; /* can this mm core dump? */
   int used_hugetlb; /* used hugetlb pages? */
   mm_context_t context; /* arch-specific data */
   int core_waiters; /* thread core dump waiters */
   struct completion *core_startup_done; /* core start completion */
   struct completion core_done; /* core end completion */
   rwlock_t ioctx_list_lock; /* AIO I/O list lock */
   struct kioctx *ioctx_list; /* AIO I/O list */
   struct kioctx default_kioctx; /* AIO default I/O context */
};
```

Los campos mmap y mm_rb son diferentes estructuras de datos que contienen la misma cosa: todas las áreas de memoria (regiones de memoria) en este espacio de direcciones. El primero almacena una lista enlazada

para permitir un recorrido simple y eficiente de todos los elementos (áreas de memoria), mientras que el segundo es un árbol binario de búsqueda rojo-negro, muy apropiado para la búsqueda de un elemento dado ($O(\log_2 n)$).

Sería interesante mostrar la información asociada al campo pgd_t *pgd (PGD, page global directory), tal y como se indica en la siguiente figura, donde el tipo pgd_t es unsigned long.



Por tanto, en primer lugar, crear un módulo cargable (mmstruct) que muestre la información más relevante de la estructura mm_struct (generar un pseudo-archivo /proc/mmstruct) del proceso actual.

Áreas de memoria de un objeto se representan por un objeto de área de memoria, que se almacenan en la estructura vm_area_struct y se encuentra definida en /linux/mm.h. A las áreas de memoria, en muchas ocasiones se les llama áreas de memoria virtual o VMAs en el kernel. La estructura vm_area_struct describe una sola área de memoria sobre un intervalo contíguo en un espacio de direcciones dado. El kernel trata a cada área de memoria como un único objeto de memoria. Cada área de memoria comparte ciertas propiedades, tales como permisos y un conjunto de operaciones asociadas. De esta forma, una sola estructura VMA puede representar varios tipos de áreas de memoria, como por ejemplo, árchivos mapeados en memoria, pila modo usuario del proceso, etc. La estructura vm_area_struct tiene la siguiente estructura:

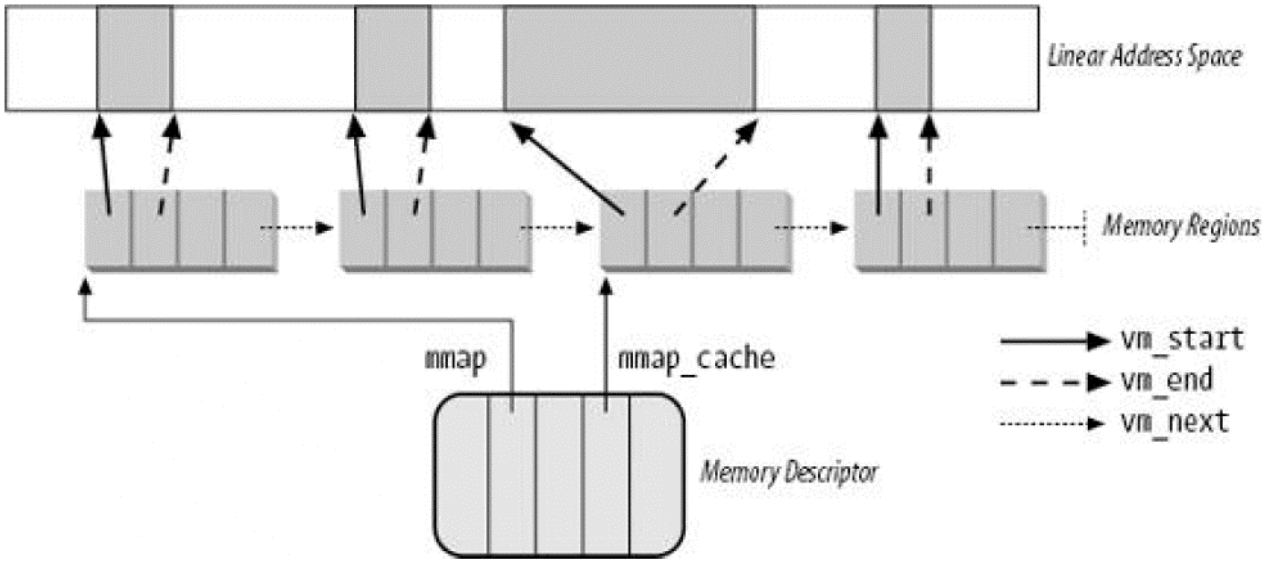
```
struct vm_area_struct {
   struct mm_struct *vm_mm; /* associated mm_struct */
   unsigned long vm_start; /* VMA start, inclusive */
   unsigned long vm_end; /* VMA end , exclusive */
   struct vm_area_struct *vm_next; /* list of VMA's */
   pgprot_t vm_page_prot; /* access permissions */
   unsigned long vm_flags; /* flags */
   struct rb_node vm_rb; /* VMA's node in the tree */
   union { /* links to address_space->i_mmap or i_mmap_nonlinear */
      struct {
         struct list_head list;
         void *parent;
         struct vm_area_struct *head;
      } vm_set;
      struct prio_tree_node prio_tree_node;
   } shared;
   struct list_head anon_vma_node; /* anon_vma entry */
   struct anon_vma *anon_vma; /* anonymous VMA object */
   struct vm_operations_struct *vm_ops; /* associated ops */
   unsigned long vm_pgoff; /* offset within file */
   struct file *vm_file; /* mapped file, if any */
   void *vm_private_data; /* private data */
};
```

Debemos destacar que cada descriptor de memoria está asociado con un único intervalo en el espacio de direcciones del proceso. El campo vm_start es la dirección inicial del intervalo y vm_end es el primer byte después de la última dirección en el intervalo. Por tanto, vm_end – vm_start es la longitud del área de memoria en bytes, que existe sobre el intervalo [vm_start, vm_end). Intervalos en diferentes áreas de memoria en el mismo espacio de direcciones no pueden solaparse.

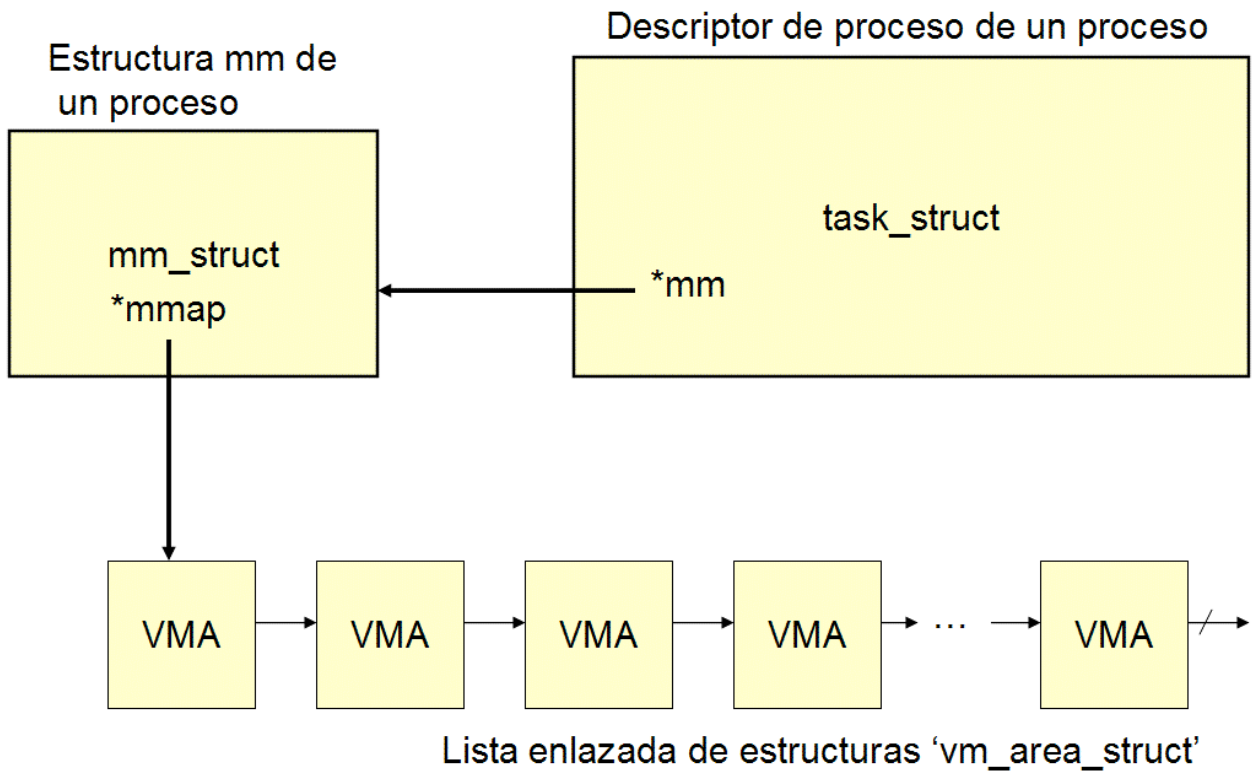
El campo vm_mm apunta al mm_struct asociado a las VMAs. Notar que cada VMA es único para el mm_struct con el que está asociado. Por tanto, incluso dos procesos independientes mapean el mismo archivo en su espacio de direcciones, cada uno tiene una única vm_area_struct para identificar su única área de memoria. Por el contrario, dos threads que comparten un espacio redirecciones también comparten todas las estructuras vm_area_struct.

El campo vm_flags contiene bit flags, definidos en linux/mm.h, que especifican el comportamiento y proporcionan información sobre las páginas contenidas en el área de memoria. A diferencia de los permisos asociados con una página física específica, los flags VMA especifican el comportamiento por el que el kernel es responsable, no del hardware. Además, vm_flags contiene información que relaciona cada página con el área de memoria, o el área de memoria como un todo, y no especifica páginas individuales. Una lista de los posibles valores de vm_flags es la siguiente: VM_READ (páginas que pueden ser leidas desde), VM_WRITE (páginas que pueden ser escritas a), VM_EXEC (páginas que pueden ser ejecutadas), VM_SHARED (páginas que son compartidas), VM_MAYREAD, VM_MAYWRITE, VM_MAYEXEC, VM_MAYSHARE, VM_GROWSDOWN, VM_GROWSUP, VM_SHM, VM_DENYWRITE, VM_EXECUTABLE, VM_LOCKED, VM_IO, VM_SEQ_READ, VM_RAND_READ, VM_DONTCOPY, VM_DONTEXPAND, VM_RESERVED, VM_ACCOUNT, VM_HUGETLB, VM_NONLINEAR, etc.

En la siguiente figura podemos observar los descriptores relacionados con el espacio de direcciones de un proceso.



En esta última figura podemos ver la relación entre los tres principales estructuras que vamos a estudiar en este módulo, que son: task_struct, mm_struct y vm_area_struct.



En primer lugar, crear un módulo cargable (vmareastruct) que muestre la información todas las VMAs que están asociados con el proceso actual "task_struct", que el kernel la gestiona como una lista enlazada de objetos "vm_area_struct" en el objeto "mm_struct" (generar un pseudo-archivo /proc/ vmareastruct) del proceso actual (currect).

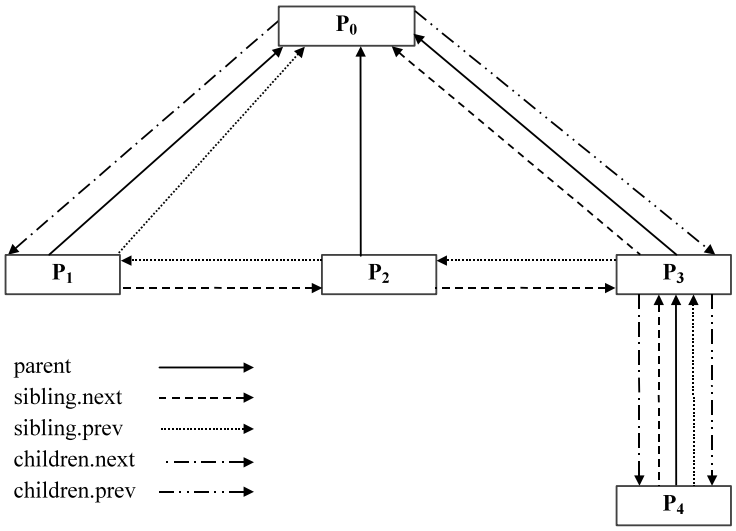
4.4. VISUALIZACIÓN DEL MAPA DE PROCESOS.

Implemente un módulo cargable *hijos.c* que debe mostrar en pantalla (a través de un pseudo-archivo *cat /proc/hijos*) el número de procesos hijo que tiene cada uno de los ancestros del proceso actual *(current)*, devolviendo también al final el total de todos ellos. Es decir, debe mostrar el número de procesos hijo que tiene cada uno de los procesos (según su *pid*) que hay que atravesar desde el proceso invocante *(current)* hasta alcanzar el proceso *init* en la jerarquía de procesos, siguiendo el puntero *parent*, y la suma de todos ellos. Indique también cómo se cargaría y cómo se extraería dicho módulo.

Todos los procesos del sistema tienen asociada, dentro del kernel, una estructura que contiene toda la información relevante de dicho proceso **struct task_struct** (descriptor de proceso) en el archivo cabecera *include/linux/sched.h*. Se suele trabajar con punteros a la tabla de procesos, luego manejaremos a menudo variables de tipo puntero a descriptor de proceso (struct task struct *). La macro *current* contiene el puntero correspondiente al proceso que está actualmente en ejecución (proceso actual). El descriptor de un proceso (struct task_struct) contiene información sobre el proceso al que representa: su pid (pid), el nombre (comm), el identificador de usuario, uid, el identificador de grupo, gid, los descriptores de archivos abiertos, y un largo etc. Entre toda la información que contiene cabe destacar los siguientes campos que serán útiles para implementar el módulo:

```
557   pid_t pid;
559   /*
560         * pointers to (original) parent process, youngest child, younger sibling,
561         * older sibling, respectively. (p->father can be replaced with
562         * p->parent->pid)
563   */
564   struct task_struct *real_parent;  /* real parent process (when being debugged) */
565   struct task_struct *parent;               /* parent process */
566   /*
567         * children/sibling forms the list of my children plus the
568         * tasks I'm ptracing.
569   */
570   struct list_head children;          /* list of my children */
571   struct list_head sibling;           /* linkage in my parent's children list */
```

El significado de los campos en el kernel de Linux 2.6 es el siguiente: *pid* (el PID del proceso); *real_parent* (puntero que apunta al descriptor del proceso que creó P o al descriptor del proceso 1 *(init)* si el proceso padre no existe. Por tanto, cuando un usuario lanza un proceso en background y cierra (exit) la shell, el proceso en background se convierte en hijo de *init*); *parent* (puntero que apunta al padre actual de P y normalmente coincide con el de *real_paren*t, salvo en determinados casos); *children* (el principio de la lista que contiene todos los hijos creados por P); *sibling* (los punteros a los elementos siguiente (next) y anterior (previous) en la lista de procesos *hermanos*, aquellos que tienen el mismo padre P. A los procesos que son todos hijos del mismo padre, se les denomina siblings (hermanos), es decir, cuando un proceso creó varios hijos (children), esos hijos están bajo la relación sibling). En la siguiente figura se pueden observar las relaciones (filiaciones) *parent* y *sibling* (en el kernel de Linux 2.6) de un grupo de procesos en el que un proceso P_0 ha creado sucesivamente tres procesos P_1, P_2 y P_3, y además el proceso P_3 ha creado al proceso P_4.




```
/* Iterar sobre hijos de un proceso */
struct task_struct *task, *task1;
struct list_head *list;
task = current;
list_for_each(list, &task->children) {
        task1 = list_entry(list, struct task_struct, sibling);
        /* task1 ahora apunta a uno de los hijos del task, para poder obtener información */
}
```

Aquí *sibling* (*children*) no es más que un puntero a la lista (*list_head*) de los procesos hermanos de un proceso P, es decir procesos que tienen como mismo padre a P (puntero a la lista de procesos creados por P). Estos punteros también están declarados en el archivo cabecera *include/linux/sched.h*.

```
570    struct list_head children;          /* list of my children */
571    struct list_head sibling;           /* linkage in my parent's children list */
```

Por otro lado, las macros *list_for_each* y *list_entry* tienen los siguientes significados: *list_for_each(p, h)* ⇒ recorre los elementos de la lista especificada por la dirección *h* del frente (cabeza) de la lista, y en cada iteración, devuelve un puntero *p* a la estructura *list_head* de la lista de elementos. *list_entry(p, t, m)* ⇒ Devuelve la dirección de la estructura de datos de tipo *t* en la que el campo *list_head* incluido en *p*, tiene el mismo nombre que *m*. Es decir, devuelve la dirección de una estructura de datos tipo *t*, de una lista *list_head* apuntada por *p*, cuyo nombre es *m*.

Una posible salida del módulo cargable podría ser la siguiente:

```
insmod (14140) -> 0 hijos
bash (14136) -> 1 hijos
su (14133) -> 1 hijos
bash (14128) -> 1 hijos
konsole (14127) -> 1 hijos
kdeinit (10083) -> 6 hijos
init (1) -> 49 hijos
Total Hijos: 59
```

Indique también cómo modificaría el módulo anterior (y el ejemplo) para que permitiese que se le introduzca un PID como parámetro.

Por último, indique qué tendría que modificar en módulo para que, para cada proceso, escriba información relacionada con los procesos hijo. Por ejemplo,

insmod (14149):
Total de 0 hijos

bash (14136):
    Hijo 1: insmod (14149)
Total de 1 hijos

su (14133):
    Hijo 1: bash (14136)
Total de 1 hijos

bash (14128):
    Hijo 1: su (14133)
Total de 1 hijos

konsole (14127):
    Hijo 1: bash (14128)
Total de 1 hijos

kdeinit (10083):
    Hijo 1: klauncher (10088)
    Hijo 2: kwin (10108)
    Hijo 3: kio_file (10124)
    Hijo 4: amule (10130)
    Hijo 5: mozilla-launche (12908)
    Hijo 6: konsole (14127)
Total de 6 hijos

init (1):
    Hijo 1: ksoftirqd/0 (2)
    Hijo 2: events/0 (3)
    Hijo 3: khelper (4)
    Hijo 4: kthread (5)
    Hijo 5: kswapd0 (152)
    Hijo 6: vesafb (154)
    Hijo 7: khpsbpkt (801)
    Hijo 8: kjournald (833)
    Hijo 9: udevd (1054)
    Hijo 10: kjournald (5677)
     ...
    Hijo 47: yakuake (10120)
    Hijo 48: kio_uiserver (11846)
    Hijo 49: gconfd-2 (12924)
Total de 49 hijos

Total Hijos: 59

4.5. (OPCIONAL) EJEMPLO DE UN DRIVER DE DISPOSITIVO MODO CARÁCTER. LEDs DE UN TECLADO ESTÁNDAR

En este módulo se pretende definir e implementar un driver de dispositivo modo carácter para los tres diodos LEDs (diodos emisores de luz, *Light Emitting Diodes*) del teclado de un PC estándar. Para escribir un valor en el archivo especial de dispositivo (también conocido como device-node) "/dev/led", los LEDs pueden ser encendidos o apagados (sólo los tres bits menos significativos de cualquier valor escrito tendrá un efecto en dichos LEDs).

bit #0: LED para el control de "Scroll-Lock",
bit #1: LED para el control de "Num-Lock",
bit #2: LED para el control de "Caps-Lock".

En general, en esta práctica nos limitaremos a implementar un manejador (driver) sencillo que muestre en el visualizador de 3 LEDs los valores que se escriban en el archivo especial y que devuelva un valor cuyos bits reflejen el estado actual de los LEDs, cuando se lea del archivo.

El manejador (driver), por tanto, hará accesible los LEDs del teclado del modo habitual en UNIX, a través de un archivo especial de dispositivo que se podrá abrir desde cualquier programa (con la llamada open, por ejemplo) y leerse y escribirse con las llamadas read y write, o bien desde el mismo shell, con echo, redirecciones, etc.

4.5.1. E/S en UNIX

En esta sección vamos a hacer una breve revisión de los conceptos más importantes relacionados con la E/S en UNIX. La abstracción que ofrece UNIX de los dispositivos físicos conectados al ordenador es la de archivo especial de dispositivo. Los archivos especiales de dispositivo suelen localizarse en el directorio */dev*, aunque esto no es obligatorio.

En los dispositivos simples, como los puertos serie, terminales, teclados, etc., la interacción se limita a menudo a abrir el archivo con la llamada *open*, para lectura o escritura y volcar o leer los datos que queremos transmitir o recibir, mediante las llamadas *read* y *write*. En muchos dispositivos, sobre todo en los más sofisticados, como tarjetas de sonido, escáneres, tarjetas de vídeo, etc., es necesario interactuar de forma más específica con el hardware. Para ello se recurre a la llamada *ioctl*, que permite modificar los parámetros del dispositivo (como velocidad, paridad, bits de stop, etc. del puerto serie; bits por muestra o frecuencia de muestreo en una tarjeta de sonido, etc.), así como instruir al periférico para que realice accesos directos a memoria u otras operaciones específicas.

Los archivos especiales (recordemos que en UNIX hay tres tipos de archivo: archivos normales o "regulares", archivos especiales y directorios) ocultan una funcionalidad especial bajo la apariencia de archivos convencionales, con su ruta de acceso, sus atributos, etc.

Distinguimos dos tipos de archivos especiales de dispositivo:

- Dispositivos de bloques: Son aquellos que pueden direccionarse por bloques, es decir que proporcionan acceso aleatorio a sectores, típicamente los discos de todo tipo.
- Dispositivos de caracteres: Son los que admiten operaciones secuenciales carácter a carácter y/o pueden direccionarse a nivel de byte.

Ejercicio: Ejecute ls −l /dev y compruebe el primer carácter de los atributos de cada archivo. Una "b" indica bloques y una "c" caracteres.

Un detalle importante a tener en cuenta es que el nombre de un archivo especial no influye para nada en su relación con uno u otro dispositivo físico. Por tanto, podemos copiar o renombrar cualquier archivo de este tipo sin ningún problema.

De hecho, estos archivos no contienen información ni ocupan espacio si los duplicamos. Los únicos datos útiles que contienen son el *major number* y el *minor number*, que identifican unívocamente el dispositivo

que representan. En el listado producido por ls −l, las dos columnas anteriores a la fecha son el *major y minor number*.

- El *major number* indica el tipo de dispositivo (por ejemplo, 3 es un disco del primer IDE, 22 uno del segundo IDE, 2 es un disco flexible, etc.)
- El *minor number* indica un número de orden dentro de los dispositivos de cada tipo (p.ej., entre los discos de un IDE, el 0 es el primer disco, el 64 el segundo, el 1 es la primera partición del primer disco, el 2 la segunda, etc.).

Ejemplos de dispositivos: (En /usr/src/linux/Documentation/devices.txt puedes encontrar la descripción completa de todos los dispositivos, incluyendo sus *major y minor numbers*).

Para crear un archivo especial de dispositivo se usa la orden *mknod*, que recibe un nombre de archivo, un tipo (bloques o caracteres), un *major number* y un *minor number*.

- */dev/fd0*: Primer floppy.
- */dev/hda*: Disco master del primer IDE.
- */dev/hdd3*: Tercera partición del disco slave del segundo IDE. (Las particiones primarias van de la 1 a la 4 y las lógicas de la 5 a la 20.)
- */dev/sda2*: Segunda partición del primer disco SCSI.
- */dev/scd0*: Primer CD-ROM SCSI. (Los CD-ROM IDE se ven como discos magnéticos.)
- */dev/st0*: Primera unidad de cinta SCSI.
- */dev/tty2*: Tercera consola.
- */dev/ttyS0*: Primer puerto serie.
- */dev/ptyp5*: Sexta pseudoterminal maestra.
- */dev/ttyp5*: Sexta pseudoterminal esclava (el proceso que crea la terminal abre la maestra y el que la va a usar abre la esclava).

Otros dispositivos físicos:

- */dev/lp0*: Primer puerto paralelo.
- */dev/parport0*: Primer puerto paralelo a nivel binario.
- */dev/psaux*: Puerto de ratón PS/2.
- */dev/sg2*: Tercer dispositivo genérico SCSI (escáner, p.ej.).
- */dev/dsp0*: Primera entrada y salida de audio digital.
- */dev/mixer1*: Segundo mezclador de audio (tarjeta de sonido).
- */dev/midi3*: Cuarto puerto MIDI (instrumentos musicales).
- */dev/video0*: Primera tarjeta de captura de vídeo.
- */dev/radio1*: Segunda tarjeta de radio.
- */dev/md1*: Segundo “metadisco” (RAID).
- */dev/ppp*: Dispositivo virtual para ppp.

Dispositivos virtuales:

- */dev/loop0*: Primer archivo “loopback”. Se asocia un archivo normal para poder montarlo como un dispositivo de bloques.
- */dev/mem*: Permite acceder a la memoria física.
- */dev/kmem*: Permite acceder a la memoria virtual del kernel.
- */dev/null*: Como un agujero negro que se lo “traga” todo.
- */dev/port*: Permite acceder a los puertos de E/S.
- */dev/zero*: Suministra incansablemente caracteres “\0”.
- */dev/full*: Devuelve error de dispositivo lleno.
- */dev/random*: Da auténticos números aleatorios de un pozo de entropía.
- */dev/urandom*: Devuelve siempre números (pseudo) aleatorios.
- */dev/stdin*: Entrada estándar del proceso en curso.
- */dev/stdout*: Salida estándar del proceso en curso.
- */dev/stderr*: Salida estándar de error del proceso en curso.

4.5.2. Programación del Driver

Para programar el driver de dispositivo modo carácter vamos a considerar las siguientes subseccione para guiar paso a paso este proceso.

4.5.2.1. Acceso a los Puertos del PC

Las rutinas para acceder (leer/escribir) a los puertos de E/S residen en include/asm−i386/io.h y son macros (no en necesario enlazar ninguna librería), de modo que basta con incluir el archivo para poder usarlas, sin necesidad de enlazar con librería alguna. Debido a una peculiaridad del compilador *gcc*, la optimización debe estar activada cuando se usan estas macros.

Para acceder a los puertos desde un programa C fuera del kernel, hay que ser superusuario y, además, requerir del sistema operativo el permiso para hacerlo. Para ello se dispone de la función *ioperm()* o *iopl()*. Desde dentro del kernel (o de un módulo), no hace falta llamar a estas funciones.

La E/S propiamente dicha se realiza a través de la función *inb(port)*, que devuelve el valor (de 8 bits) al que se encuentra el puerto correspondiente, y de la función *outb(valor, puerto)*, que vuelca un valor de 8 bits sobre un puerto.

4.5.2.2. Reserva de Puertos

Desde el kernel de Linux, el acceso a los puertos de E/S está regulado mediante un mecanismo de reserva de rangos de direcciones. Esto permite que los manejadores sepan si un rango de puertos está siendo utilizado por otros manejadores. Para ello se dispone de tres funciones que debemos emplear para no colisionar con otros manejadores. Podemos saber en un momento dado cuáles son los rangos de puertos reservados por los diferentes manejadores que se hallan instalados en el sistema visualizando el archivo */proc/ioports*.

- *int check_region(unsigned int port, unsigned int range)*: Comprueba si un rango de *range* puertos a partir de *port* están siendo utilizadas por otro módulo, devolviendo un valor negativo si es así. *kernel/resource.c*.
- *void request_region(unsigned int port, unsigned int range, const char *name)*: Reserva un rango de *range* puertos a partir de *port*, identificándolo con *name*. *kernel/resource.c*.
- *void release_region(unsigned int port, unsigned int range)*: Libera un rango de *range* puertos a partir de *port*, reservados anteriormente. *kernel/resource.c*.

4.5.2.3. Registro del Driver en el Sistema

En esta práctica, el módulo que queremos desarrollar ha de controlar un *dispositivo de caracteres*, es decir que el kernel debe saber que cuando un proceso acceda a cierto archivo especial (que tendrá asignado un *major number* concreto), debe enviar la petición correspondiente a nuestro código. Para notificar esto al kernel, hemos de llamar nada más insertar el módulo, a la función:

*register_chrdev(unsigned int major, const char *name, struct file_operations *fops)*,

que se encuentra definida en *fs/devices.c*, y que recibe como parámetros el *major number* al queremos asociar el driver, un nombre *(name)* que identifique al dispositivo (el nombre que le vamos a dar al driver) y una estructura *file_operations* que indicará al kernel qué función tiene que invocar para cada tipo de acceso al dispositivo (una para open(), otra para close(), etc.).

Si se registra correctamente el *major number*, *register_chrdev()* nos devuelve un 0 o mayor, sino, un código de error negativo. Utilizaremos en esta práctica el *major number* 55, que no está utilizado en las computadoras del laboratorio (corresponde al dispositivo /dev/dsp56k, que es una placa con el procesador digital de señal DSP56001), aunque también se puede utilizar el *major number* 0 (unnamed for NFS, network, ec.). Como puede comprobar, a la función *register_chrdev()* no se pasa ningún *minor number*, ya que en realidad el sistema no emplea dicho número para nada, y lo pasa directamente al manejador cuando hay una operación sobre uno de ellos.

Para crear un archivo especial de caracteres que tenga asociado este número, se utilizará la orden *mknod*. El archivo especial puede llamarse, /dev/led, aunque cualquier otro nombre de archivo es igualmente válido. Acuérdese de borrar el archivo al terminar cada sesión.

4.5.2.4. Resumen del Proceso de Carga/Descarga del Driver

Para realizar el proceso de carga y descarga del driver habrá que tener en cuenta las siguiente operaciones:
Al arrancar el módulo, en *init_module*:

- Comprobar la disponibilidad del rango de puertos de E/S que vayamos a utilizar y reservarlo.
- Registrar el driver con la función *register_chrdev()*;
- Borrar (si lo hubiese) el archivo especial de dispositivo anterior. Utilizando la función *set_fs()* y la llamada al sistema *sys_unlink()*.
- Crear un nuevo archivo especial de dispositivo, utilizando la llamada al sistema *sys_mknod()*.
- Cambiar los permisos por defecto con los que se crea el archivo espacial de dispositivo, utilizando para ello la llamada al sistema sys_chmod() y mapeando el permiso 0666, para que todo el mundo pueda leer y escribir.

Al descargar el módulo, en *cleanup_module*:

- Deregistrar el driver. Esta operación la lleva a cabo la función *unregister_chrdev(unsigned int major, const char * name)*, que se encuentra en *fs/devices.c*, a la que hay que pasarle el major number y el nombre que se utilizó al registrarse.
- Liberar también el rango de puertos de E/S que se reservó al principio.

4.5.2.5. Funciones de Manejo del Dispositivo

Vamos a examinar las operaciones sobre dispositivos y cómo se indica al kernel qué funciones llamar para cada operación. El último parámetro que se le pasa a la función *register_chrdev()* es struct file_operations que se encuentra en *include/linux/fs.h*. Los campos de esta estructura indican al kernel qué funciones llamar cuando se producen las operaciones de apertura (open()), cierre (close()), lectura (read()), escritura (write()), etc. sobre un archivo especial de dispositivo que tenga el *major number* correspondiente. En nuestro caso, sólo implementaremos las funciones básicas (existen algunas otras, como select(), ioctl()), etc., con lo cual la estructura apuntada por fops puede ser la siguiente:

```
struct file_operations led_fops =
{
        owner:      THIS_MODULE,
        read:       led_read,
        write:      led_write,
        open:       led_open,
        release:    led_release,
};
```

Vamos a comentar a continuación la implementación de las cuatro funciones de manejo del dispositivo que necesitamos para esta práctica:

- *static int led_open(struct inode *inode, struct file *file)*: Debe hacer un *MOD_INC_USE_COUNT* para indicar que se está usando el módulo, ya que se ha abierto un dispositivo que éste está controlando. Esta macro incrementa un contador asociado al módulo, lo que impedirá que el módulo pueda eliminarse del sistema con *rmmod*. Si no se han hecho tantas *MOD_DEC_USE_COUNT* como *MOD_INC_USE_COUNT*, el código del kernel que sirve la llamada de *rmmod* (extraer módulo) dará un mensaje de "módulo en uso". La función debe devolver un cero para indicar que se ha abierto el dispositivo con éxito.
- *static int led_release(struct inode *inode, struct file *file)*: Debe hacer un *MOD_DEC_USE_COUNT* para decrementar el contador de dispositivos abiertos (en su caso, podría haber varios con distintos minor numbers). Cuando ese contador vale cero, se puede extraer el módulo con *rmmod*. Debe devolver 0 si todo ha ido bien.

- *static ssize_t led_read(struct file *file, char *buffer, size_t count, loff_t *pos)*: Debe hacer un *inb()* del puerto correspondiente para leer el estado de los tres LEDs y devolverlo como una cadena de 1 byte en buf. Devolverá como resultado de la función el número de bytes leídos (siempre 1).
- *static ssize_t led_write(struct file *filp, const char *buf, size_t count, loff_t *pos)*: Debe hacer un *outb()* al puerto de datos correspondientes a los tres bits menos significativos del carácter de buf. Devolverá como resultado de la función el número de bytes escritos (siempre 1). En general el algoritmo para establecer los LEDs constaría de los siguientes pasos: **(1)** bloquear interrupciones para entrar en la "sección crítica" (cli() para no reconocer interrupciones); **(2)** esperar hasta que el buffer de entrada del controlador de teclado esté lleno; **(3)** enviar como salida el byte-orden "LED_WRITE" controlador del teclado; **(4)** esperar hasta que el controlador ha aceptado este comando; **(5)** enviar como salida el byte-indicador LED al controlador de teclado; **(6)** esperar hasta que el controlador haya aceptado este byte-dato; **(7)** permitir las interrupciones al salir de la "sección crítica" (sti() para reconocer interrupciones).

4.5.3. Ejemplo Sencillo de un Driver

En esta sección mostraremos el más sencillo de los driver de dispositivo modo carácter que se puede desarrollar, más aún, éste es sólo un ejemplo de demostración que no implementa ninguna operación.

```
#define MODULE
#define __KERNEL__

#include <linux/module.h>
#include <linux/kernel.h>
#include <asm/unistd.h>
#include <sys/syscall.h>
#include <sys/types.h>
#include <asm/fcntl.h>
#include <asm/errno.h>
#include <linux/types.h>
#include <linux/dirent.h>
#include <sys/mman.h>
#include <linux/string.h>
#include <linux/fs.h>
#include <linux/malloc.h>

/* Prototipos de funciones de servicio del driver */
static int driver_open(struct inode *, struct file *);

/* Registra cada función que será suministrada por el driver a los usuarios que lo utilicen */
static struct file_operations driver_fops = {
        owner:      THIS_MODULE,
        open:       driver_open,
};

/* Función de inicialización del módulo */
int init_module(void)
{
        /* Registrar el driver con el major number 40 y el nombre "driver" */
        if (register_chrdev(40, "driver", &driver_fops))
                return -EIO;
        return 0;
}

/* Función de descarga del módulo */
void cleanup_module(void)
{
```

```
        /* Deregistra el driver */
        unregister_chrdev(40, “driver”);
}

/* Esto es solo una prueba de demostración (la función imprime sólo un mensaje) */
static int driver_open(struct inode *inode, struct file *file)
{
        printk(“<1> Función que ‘open’ el driver\n”);
        return 0;
}
```

En este ejemplo, en realidad la función más importante es *register_chrdev()*, que registra el driver con el major number 40. Si se desea acceder a este driver tenemos que hacer lo siguiente:
mknod /dev/driver c 40 0
insmod driver.o

Después de la ejecución de estas órdenes se puede utilizar el driver, aunque debe tener en cuenta que no implementa ninguna función que haga realmente algo (sólo open), aunque el principal objetivo es mostrar el esqueleto que tiene la implementación de un driver de dispositivo modo carácter en Linux.

4.5.4. Comprobación del Funcionamiento

Para comprobar el funcionamiento de los tres LEDs del teclado (driver a implementar en la práctica) puede volcar cualquier archivo sobre el dispositivo con: # cat archivo > /dev/led. El problema es que la visualización será tan rápida que sólo aparecerá visible el último carácter válido del archivo.

Por otro lado, para comprobar la lectura de los LEDS, bastará con hacer, # cat /dev/led, y veremos aparecer continuamente la representación de ceros y unos correspondiente a la posición actual de los LEDs. Para parar, bastará con pulsar Ctrl−c.

También es posible realizar un programa C que pruebe el funcionamiento del módulo implementado en esta parte de la práctica.

BIBLIOGRAFÍA

[LKD05] **Linux Kernel Development**. Robert Love (2005).
[ULK05] **Understanding Linux Kernel**. Daniel P. Bovet y Marco Cesati (2005)
[LDD05] **Linux Device Drivers**. Jonathan Corbet, Alessandro Rubini y Greg Kroah-Hartman (2005)