LECCIÓN 1: INTRODUCCIÓN AL SISTEMA OPERATIVO LINUX

LECCIÓN 1: INTRODUCCIÓN AL SISTEMA OPERATIVO LINUX....1
LECCIÓN 1: INTRODUCCIÓN AL SISTEMA OPERATIVO LINUX....1
1.1 Génesis....1
1.2 Arquitectura del núcleo....2
1.3 Anatomia del núcleo....4
1.4 Recorrido por el directorio de los fuentes de Linux....7
1.5 Utilización de una función del sistema operativo por un usuario....11
Llamada al sistema: open....11

LECCIÓN 1: INTRODUCCIÓN AL SISTEMA OPERATIVO LINUX

1.1 Génesis

El sistema operativo Linux se genera inspirándose en dos sistemas operativos, el sistema abierto UNIX creado en 1969 por Ken Thompson y Dennis Ritchie en los laboratorios de Bell. De este sistema se toman sus características, especificaciones y funcionamiento. Mas el sistema educativo Minix creado en 1987 por Andrew S. Tanenbaum del cual se toma la estructura y código del núcleo. Con todo esto en 1991 Linus Torvalds crea Linus's Unix = Linux Kernel, esto es crea solo el núcleo del sistema sin la capa de servidores, manejadores, aplicaciones graficas, etc que serán creadas posteriormente por otros autores. El código del núcleo lo podemos encontrar en la dirección (www.kernel.org). El núcleo actual tiene aproximadamente 1,5 millones de líneas de código, y representa menos del 50 por ciento de todo el código del sistema.

En la comunidad de programadores se crea el proyecto GNU (Gnu's Not Unix), proyecto para generar software libre, donde se generan editores Emacs, compiladores gcc, interprete de comandos bsh, sistema operativo Hurd, aplicaciones, etc., bajo la licencia publica general GPL (General Public License), usar, copiar, distribuir y modificar (con las mismas condiciones). Se conserva la firma del autor. Se puede cobrar.

Linux se crea con esta filosofía de libre distribución y el sistema operativo completo que se construye con este núcleo también. A todo el sistema se le da el nombre de GNU/Linux (distribución completa del sistema operativo con Linux), que contiene el núcleo (1,5 millones de líneas de código) mas las otras capas del sistema operativo y utilidades. Si bien muchas veces se denomina a todo el sistema simplemente LINUX.

CARACTERISTICAS DEL NÚCLEO

- Su código es de libre uso.
- Escrito en lenguaje C, compilado con gcc, primera capa en ensamblador.
- Ejecutable en varias plataformas hardware.
- Se ejecuta en máquinas con arquitectura de 32 bits y 64 bits.
- Estaciones de trabajo y servidores.
- Código y funcionamiento escrito bajo la familia de estándares POSIX (Portable Operating System Interface).
- Soporta CPU's con uno o varios microprocesadores (SMP) symmetric multiprocessing.
- Multitarea.
- Multiusuario.
- Gestión y protección de memoria.

- Memoria virtual.
- Varios sistemas de ficheros.
- Comunicación entre procesos (señales, pipes, IPC, sockets).
- Librerías compartidas y dinámicas.
- Permite trabajar en red TCP/IP.
- Soporte grafico para interfase con el usuario.
- Estable, veloz, completo y rendimiento aceptable.
- Funcionalmente es muy parecido a UNIX.
- Mas de 100.000 usuarios.
- Actualizado, mejorado, mantenido y ampliado por la comunidad de usuarios (modelo “bazar”, contrapuesto al modelo “catedral”).

1.2 Arquitectura del núcleo

Veamos los objetivos mas importantes que se han tenido en cuenta a la hora de diseñar el núcleo, basados en claridad, compatibilidad, portabilidad, robusto, seguro y veloz.

CLARO
Claridad y velocidad son dos objetivos normalmente contrapuestos. Los diseñadores suelen sacrificar claridad por velocidad. La claridad complementa la robustez del sistema y facilita la realización de cambios y mejoras. Podemos encontrar reglas de estilo en el fichero /usr/src/Documentation/CodingStyle.

COMPATIBLE
Diseñado bajo la normativa POSIX. Soporta ejecutar ficheros Java. Ejecuta aplicaciones DOS, con el emulador DOSEMU. Ejecuta algunas aplicaciones Windows a través del proyecto WINE. Se mantiene compatibilidad con ficheros Windows a través de los servicios SAMBA. Soporta varios sistemas de ficheros
ext2 y ext3 (sistemas de ficheros nativos),ISO-9660 usado sobre los CDROMs, MSDOS, NFS (Network File System), etc. Soporta protocolo de redes TCP/IP, soporta protocolo de AppleTalk (Macintosh), también protocolos de Novell IPX (Internetwork Packet Exange), SPX (Sequenced Packet Exange), NCP (NetWare Core Protocol), y IPv6 la nueva versión de IP. Mantiene compatibilidad con una variedad de dispositivos hardware.

MODULAR
El núcleo define una interfaz abstracta a los subsistemas de ficheros denominada VFS (Virtual File System Interface), que permite integrar en el sistema varios sistemas de ficheros reales. También mantiene una interfase abstracta para manejadores binarios, y permite soportar nuevos formatos de ficheros ejecutables como Java.

PORTABLE
Mantiene una separación entre código fuente dependiente de la arquitectura y código fuente independiente de la arquitectura, que le permite ejecutar Linux en diversas plataformas como Intel, Alpha, Motorola, Mips, PowerPC, Sparc,

Macintoshes, Sun, etc. Existen versiones del núcleo especificas para portatiles, y PAD's.

ROBUSTO Y SEGURO
El disponer de los fuentes, permite a la comunidad de usuarios modificar los errores detectados y mantener el sistema constantemente actualizado. Dispone de mecanismos de protección para separar y proteger los programas del sistema de los programas de los usuarios. Soporta cortafuego para protección de intrusos.

VELOZ
Este es el objetivo mas demandado por los usuarios, si bien no suele ser un objetivo crucial. El código está optimizado así encontramos la primera capa del núcleo escrita en ensamblador.

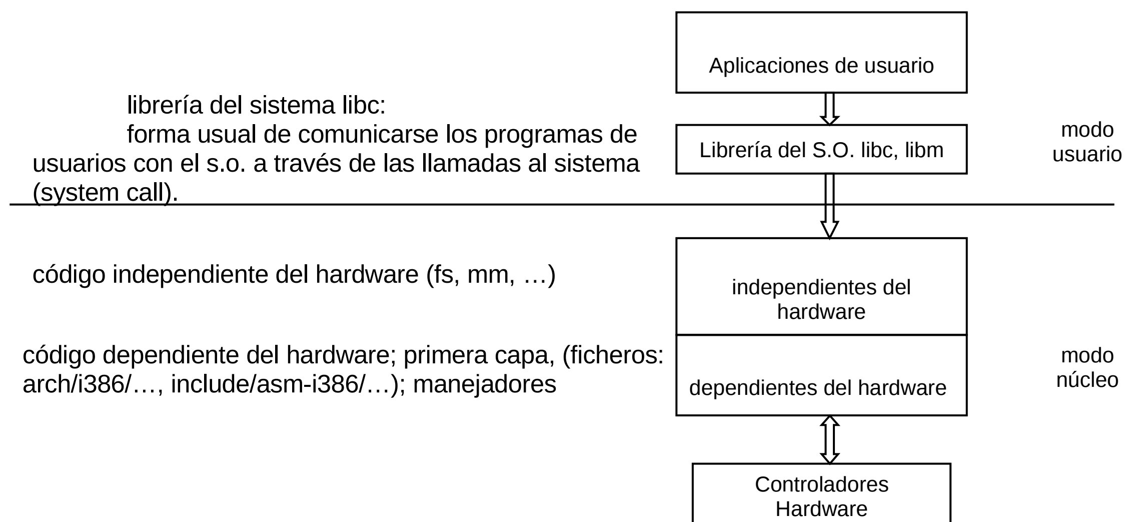
VISTA MODULAR DEL NÚCLEO

Una primera división del núcleo podemos establecerla atendiendo a:

- código dependiente del hardware; primera capa, (ficheros: arch/i386/…, include/asm-i386/…); manejadores.

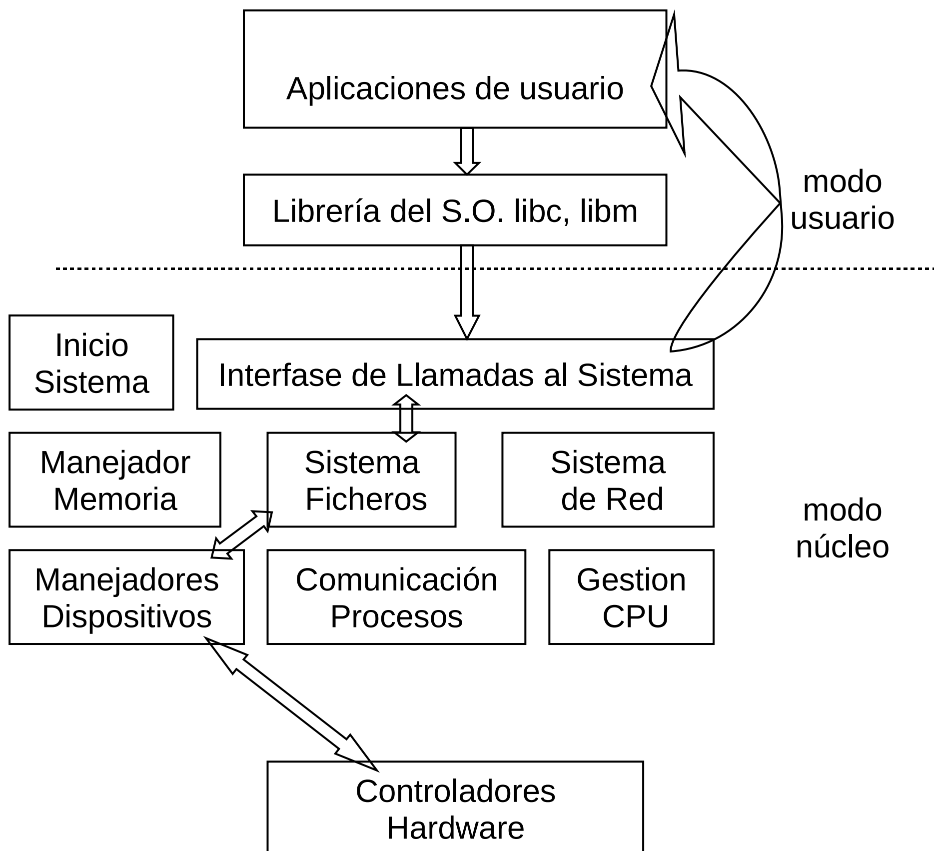
- código independiente del hardware (fs, mm, …).

La interfase entre los usuarios y el núcleo se establece a través de la librería del sistema libc, contiene los procedimientos y utilidades que permiten a los programas de usuarios comunicarse con el s.o. por medio de las llamadas al sistema (system call).



1.3 Anatomia del núcleo

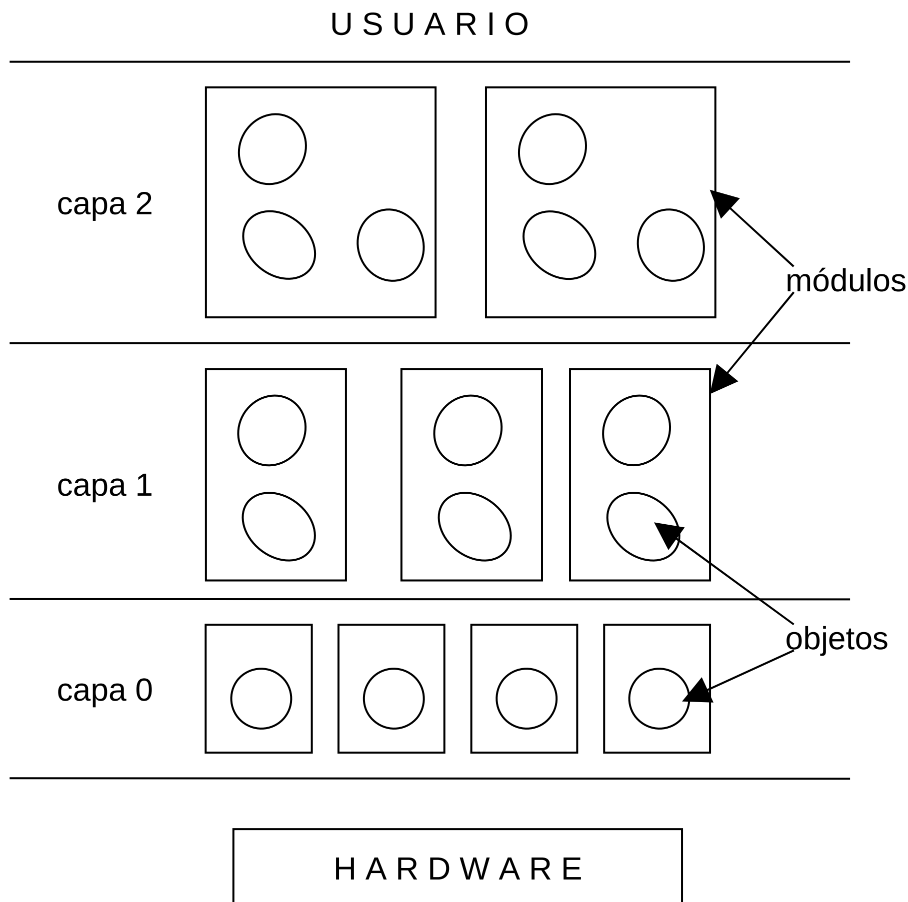
A nivel funcional



La interacción o solicitud de un servicio por parte de un programa de usario, sigue los siguientes pasos:

1. El usuario en su programa solicita una función de la librería libc.
2. la función de libc, hace una llamada al núcleo a través de una llamada (syscall).
3. el núcleo recibe esa llamada mediante una función (system call).
4. el núcleo redirige esta llamada a los módulos implicados que a su vez llaman a otros hasta llegar al hardware y realizar la solicitud del usuario.
5. el núcleo hace llegar a la aplicación del usuario el resultado de su solicitud.

CAPAS, MODULOS, OBJETOS, COMPONENTES



CAPAS

- Cada capa proporciona una serie de funciones a las capas superiores.
- Las funciones en cada capa se construyen con recursos de la propia capa o con funciones de capas inferiores.
- Cada capa tiene su lenguaje de descripción.
- El lenguaje de descripción y uso de los recursos aumenta de nivel a medida que subimos hacia el usuario.
- En el nivel mas bajo el nivel de descripción es el nivel maquina: in, out, lock, …, es el que entienden los controladores hardware.
- En el nivel mas alto o de usuario el lenguaje de descripción se asemeja al lenguaje natural: imprimir, ▯ icono de una impresora, ● voz “imprimir”, …
- Se pueden implementar mecanismos de seguridad por capas.

MÓDULOS

- Ofrecen funciones.

- Separan interfase de uso de su implementación.
- Ocultan a los usuarios aspectos de implementación.
- Se pueden cambiar módulos sin afectar a otros módulos o al sistema.

OBJETOS – COMPONENTES

- Es una forma más estructurada de implementar los módulos.
- Se encapsulan estructuras de datos con los procedimientos que las manejan.
- Componentes un refinamiento del concepto de objetos.
- Linux implementa una estructura por capas con módulos.
- No esta implementada una estructura de objetos o componentes.

ESTRUCTURA DE MICRONÚCLEO (Minix, Mach)

- El núcleo se implementa mediante varios procesos o módulos separados núcleo, mm, fs, net, …
- Los procesos se ejecutan en modo privilegiado y se comunican mediante mensajes.
- Tiene ventajas en el diseño y en la actualización de un modulo.
- Módulos que no se necesitan no tienen que ser cargados.
- Desventaja, la utilización de un recurso de otro modulo se solicita por mensajes lo que lo hace mas lento.

ESTRUCTURA MONOLÍTICA O MACRONÚCLEO (Unix)

- El núcleo es un único gran proceso.
- La utilización de un procedimiento se llama directamente, no necesita mensajes, por lo que es más rápido.
- Actualizaciones, implican recompilar todo el núcleo.

ESTRUCTURA DE LINUX

- Linux mantiene una estructura monolítica, pero admite módulos cargables.

- Módulos pueden ser manejadores de dispositivos, y se cargan cuando se necesitan.

1.4 Recorrido por el directorio de los fuentes de Linux

Los fuentes de Linux, se encuentran en el directorio /usr/src/linux estructurados en varios subdirectorios. Aproximadamente 2 millones de líneas de código.

- documentation
  documentación relativa al núcleo.

- arch (arch-itecture)
  en este subdirectorio encontramos código dependiente de la arquitectura (400.000 líneas de código, 25 % del total del código del núcleo).

- arch/alpha, arch/m68k, arch/mips, arch/sparc, …

- arch/i386/
  código para la arquitectura Intel y compatibles (AMD, Cyrix, IDT) (50.000 líneas de código) contiene los subdirectorios:

  - arch/i386/kernel
    manejo de señales, SMP (simetric multiprocesing).

  - arch/i386/lib
    librería rápida genérica con funciones como strlen, memcpy.

  - arch/i386/mm
    procedimientos del manejador de memoria dependientes de la arquitectura.

- drivers
  contiene los fuentes de los manejadores de los dispositivos hardware, (64.000 líneas de código para una arquitectura). Contiene subdirectorios.
  - cha
  - block
  - cdrom

- fs
  contienen subdirectorios con todos los sistemas de fichero reales soportados por Linux.

  - ext2 para dispositivos localizados en el equipo.

  - NFS para dispositivos accedidos a través de la red.

  - proc seudo sistema de ficheros para acceder a variables y estructuras del núcleo.

- include
  contiene ficheros cabecera .h, y se divide en subdirectorios:

- include/asm-*
  existe uno para cada arquitectura.
- include/asm-i386
  contienen macros para el preprocesador y funciones para la arquitectura, muchas de estas funciones están implementadas en lenguaje ensamblador para mayor rapidez.

- include/linux
  básicamente define constantes y estructuras del núcleo.

- include/net
  ficheros cabecera para el sistema de red.

- include/SCSI
  ficheros cabecera para el controlador SCSI.

- include/video
  ficheros cabecera para placas de video.

- init
  su principal fichero es main.c para la inicialización del núcleo.

- ipc
  ficheros para la comunicación entre procesos.

- kernel
  es la capa mas interna del núcleo que no depende de la arquitectura, contiene los procedimientos básicos como, manejo de la cpu, crear y terminar procesos, etc.

- lib
  contiene dos partes:

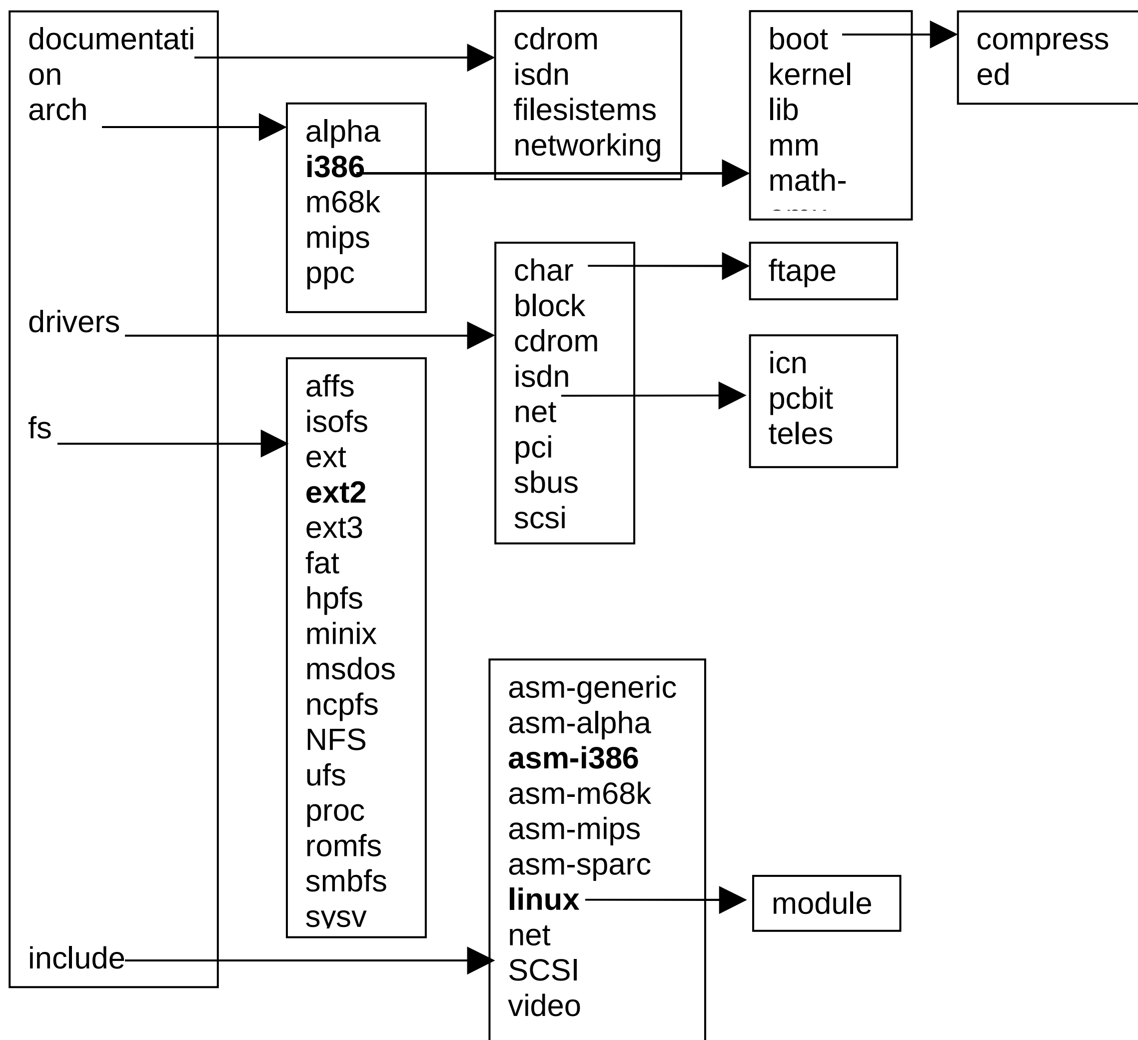
  - lib/inflate.c
    descomprime y comprime el núcleo cuando el sistema esta arrancando.

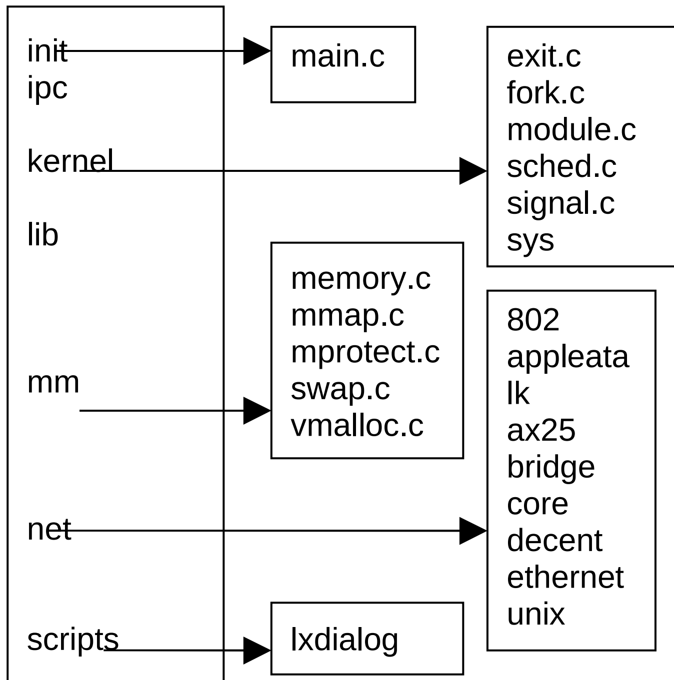
  - librería C estándar manejo de string, memoria, etc.

- mm
  contiene los procedimientos y estructuras del manejador de memoria que son independientes de la arquitectura.

- net
  contiene los procedimientos y estructuras de subsistema de red independiente de la arquitectura con los protocolos de red TCP/IP, IPX, AppleTalk, etc.

- scripts
  este subdirectorio no contiene código, contiene scritps para configurar el núcleo.


documentati
on
arch
drivers
fs
include
alpha
i386
m68k
mips
ppc
affs
isofs
ext
ext2
ext3
fat
hpfs
minix
msdos
ncpfs
NFS
ufs
proc
romfs
smbfs
sysv
cdrom
isdn
filesistems
networking
char
block
cdrom
isdn
net
pci
sbus
scsi
asm-generic
asm-alpha
asm-i386
asm-m68k
asm-mips
asm-sparc
linux
net
SCSI
video
boot
kernel
lib
mm
math-
compress
ed
ftape
icn
pcbit
teles
module


init
ipc
kernel
lib
mm
net
scripts
main.c
exit.c
fork.c
module.c
sched.c
signal.c
sys
memory.c
mmap.c
mprotect.c
swap.c
vmalloc.c
802
appleata
lk
ax25
bridge
core
decent
ethernet
unix
lxdialog

1.5 Utilización de una función del sistema operativo por un usuario

Llamada al sistema: open

Los usuarios utilizan las funciones del sistema operativo mediante las llamadas al sistema (~200 en versión 2 de Linux), definidas en el sistema por un nombre y un número, (open, 5).

El nombre de la llamada se encuentra definido en el archivo ensamblador arch/i386/kernel/syscall_table.S

```
.long sys_open        /* 5 */
```

El número en include/asm-i386/unistd.h

```
#define  NR_open   5
```

1. Un usuario en su programa en C invoca una función de la biblioteca del sistema
   fd = open ("/dev/hda", O_RDONLY);

2. La biblioteca en C, (libc, libm) proporciona funciones que invocan a las llamadas al sistema open -> syscall -> INT 0x80 (parámetros de la llamada en los registros del procesador).
INTERRUPCIÓN SOFTWARE paso de modo usario a modo NÚCLEO .

3. mediante las tablas IDT + GDT se ejecuta la función sys_call definida en arch/i386/kernel/entry.s
   sys_call utiliza el número de llamada al sistema (open, 5) transmitido en el registro eax para buscar en la tabla sys_call_table la dirección de la llamada al sistema.

4. las llamadas tienen la sintaxis: sys_nombre (argumentos)
   int sys_open (const char *filename, int flags, int mode).

5. cuando la llamada termina vuelve el control a sys_call y este devuelve el control al usuario y se retorna a modo USUARIO.

Programa para leer el primer sector del disco duro. (según la idea de Andries Brouwer, aeb@cwi.nl)

- Abre el fichero especial “/dev/hda”, asociado al primer disco duro IDE.
- Lee en la variable “buf” el primer sector del disco

```
#include <unistd.h> /* números de las llamadas al sistema */
#include <fcntl.h>
int main(){
        int fd;
```

```
        char buf[512];
        fd = open("/dev/hda", O_RDONLY);
        if (fd >= 0)
              read(fd, buf, sizeof(buf));
        return 0;
}
```

open

- El fuente de la llamada al sistema sys_open se encuentra en fs/open.c, las líneas maestras de la llamada son:
- (1) Obtiene memoria en el núcleo y pasa el nombre del fichero al núcleo.
- (2) Obtiene un descriptor de fichero no utilizado fd.
- (3) Realiza el open, busca el inode asociado al fichero.
- (4) Almacena información del paso (3) en la ranura del programa que llama.
- (5) Libera el espacio de memoria en el núcleo reservada en (1) devuelve el descriptor de fichero fd

sys_open

- Veamos los fuentes, quitando las líneas no esenciales, el manejo de errores y de bloqueo.

```
int sys_open(const char *filename, int flags, int mode) {

      char *tmp = getname(filename);                          /*(1)*/

      int fd = get_unused_fd();                               /*(2)*/

      struct file *f = filp_open(tmp, flags, mode);           /*(3)*/

      fd_install(fd, f);                                      /*(4)*/

      putname(tmp);                                           /*(5)*/

     return fd;
}
```

(1) getname()

- La función getname() se encuentra en fs/namei.c. Pasa parámetros al núcleo.
- asigna memoria en el núcleo para el nombre del fichero “/dev/hda” .

- copia el nombre del fichero desde el espacio del usuario al espacio del núcleo.

```
#define __getname()     kmem_cache_alloc(names_cachep, SLAB_KERNEL)
#define putname(name)   kmem_cache_free(names_cachep, (void *)(name))

char *getname(const char *filename) {
        char *tmp = __getname();        /* asigna memoria */

        strncpy_from_user(tmp, filename, PATH_MAX + 1);

        return tmp;}
```

(2) get_unused_fd()

- El procedimiento get_unused_fd() se encuentra en fs/open.c.
- current es un puntero que señala en la tabla de procesos (task struct) la ranura del proceso que hizo el open.
- la estructura files almacena la información de ficheros del proceso busca el primer descriptor de fichero libre
- lo marca como usado en la estructura files
- devuelve el descriptor de fichero encontrado

```
int get_unused_fd(void) {
        struct files_struct *files = current->files;
        int fd = find_next_zero_bit(files->open_fds,
                                    files->max_fdset, files->next_fd);
        FD_SET(fd, files->open_fds);    /* en uso ahora */
       files->next_fd = fd + 1;
        return fd;
}
```

(4) fd_install()

- El procedimiento fd_install() se encuentra en include/linux/file.h.

- Almacena información devuelta por filp_open() en la estructura files.

```
void fd_install(unsigned int fd, struct file *file) {

       struct files_struct *files = current->files;

        files->fd[fd] = file;
}
```

(3) filp_open()

- La function filp_open() se encuentra en fs/open.c.

- Realiza el trabajo principal de sys_open().
- El inode contiene la información que precisa el s.o. sobre un fichero.
- dentry, entrada a un directorio describe el nombre de un fichero, mediante el número del inode mas la ruta para encontrarlo.
- Utiliza la estructura nameidata que se encuentra definida en include/linux/fs.h. en la búsqueda del inode, almacena la entrada al directorio dentry y el sistema de fichero utilizado mnt que nos lleva a los procedimientos que manejan las estructuras, superbloque, inode, … de ese FS.

```
struct nameidata {
        struct dentry *dentry; /*entrada en el directorio*/
        struct vfsmount *mnt; /*tipo de sistema de ficheros utilizado*/
        struct qstr last; /* contiene el path */
};
```

filp_open()

- busca el inode asociado con el fichero mediante open_namei
- busca una entrada en la tabla filp, con dentry_open

```
struct file *filp_open(const char *filename, int flags, int mode) {
        struct nameidata nd;
        open_namei(filename, flags, mode, &nd);
        return dentry_open(nd.dentry, nd.mnt, flags);
}
```

open_namei()

- El procedimiento open_namei() se encuentra en fs/namei.c:
- llena campos de la estructura nameidata (nd->mnt y nd->dentry).
- realiza la búsqueda de la entrada de directorio mediante path_walk() .

```
open_namei(const char *pathname, int flag, int mode, struct nameidata *nd) {
        if (!(flag & O_CREAT)) {
                /* El caso mas simple – realizar una búsqueda sencilla. */
                if (*pathname == '/') {         /*directorio root*/
                        nd->mnt = mntget(current->fs->rootmnt);
                nd->dentry = dget(current->fs->root);
                } else {                        /*directorio actual*/
                        nd->mnt = mntget(current->fs->pwdmnt);
                nd->dentry = dget(current->fs->pwd);
                }
                path_walk(pathname, nd);
                /* Check permissions etc. */
                ...
                return 0;
        }
```

```
        ...
}
```

path_walk

- Busca en la cache de directorios recientemente usados la entrada de directorio mediante cached_lookup(). Si no encuentra la entrada.
- Busca la entrada de directorio en el sistema de ficheros mediante real_lookup(), que irá a disco.
- Al finalizar nd contiene la entrada al directorio, o lo que es lo mismo el inode con la información del fichero.

```
path_walk (const char *name, struct nameidata *nd) {
        struct dentry *dentry;
        for(;;) {
                struct qstr this;
                this.name = next_part_of(name);
                this.len = length_of(this.name);
                this.hash = hash_fn(this.name);
                /* if . or .. then special, otherwise: */
                dentry = cached_lookup(nd->dentry, &this);
                if (!dentry)
                        dentry = real_lookup(nd->dentry, &this);
                nd->dentry = dentry;
                if (this_was_the_final_part)
                        return;
        }
}
```

dentry_open()

- encuentra una entrada vacía en la tabla “filp”.
- initializa la estructura “file”.

```
struct file *

dentry_open(struct dentry *dentry, struct vfsmount *mnt, int flags) {
        struct file *f = get_empty_filp();
        f->f_dentry = dentry;
        f->f_vfsmnt = mnt;
        f->f_pos = 0;
        f->f_op = dentry->d_inode->i_fop;
        ...
        return f;
}
```

Procedimientos para trabajar con el FS

- Cada sistema de fichero proporciona unas estructuras que contienen los procedimientos para trabajar con el:
- super_operations, procedimientos para trabajar con la estructura superbloque.
- file_operations, procedimientos para trabajar con la estructura file. inode_operations, procedimientos para trabajar con la estructura inode.
- address_space_operations direcciones de procedimientos que hacen otras cosas y a sí mas.
- Para un sistema de ficheros especifico, la dirección de la función de búsqueda real_lookup, se encuentra en la estructura inode_operations en el inode contenido en la entrada “dentry” del directorio para el que estamos realizando la búsqueda.
- Esta función especifica para un determinado sistema de ficheros, debe leer en el disco y buscar en el directorio para este fichero que estamos buscando.

```
struct dentry *
real_lookup(struct dentry *parent, struct qstr *name, int flags) {
        struct dentry *dentry = d_alloc(parent, name);
        parent->d_inode->i_op->lookup(dir, dentry);
        return dentry;
}
```

fs/romfs/inode.c

Ejemplos de sistemas de fichero son minix y romfs porque ellos son sencillos y pequeños. Por ejemplo, en fs/romfs/inode.c:

```
romfs_lookup(struct inode *dir, struct dentry *dentry) {
        const char *name = dentry->d_name.name;
        int len = dentry->d_name.len;
        char fsname[ROMFS_MAXFN];
        struct romfs_inode ri;
        unsigned long offset = dir->i_ino & ROMFH_MASK;
        for (;;) {
                romfs_copyfrom(dir, &ri, offset, ROMFH_SIZE);
                romfs_copyfrom(dir, fsname, offset+ROMFH_SIZE, len+1);
                if (strncmp (name, fsname, len) == 0)
                        break;
                /* next entry */
                offset = ntohl(ri.next) & ROMFH_MASK;
        }
        inode = iget(dir->i_sb, offset);
        d_add (dentry, inode);
        return 0;
}
```

romfs_copyfrom()

```
romfs_copyfrom(struct inode *i, void *dest,
                unsigned long offset, unsigned long count) {
        struct buffer_head *bh;
        bh = bread(i->i_dev, offset>>ROMBSBITS, ROMBSIZE);
        memcpy(dest, ((char *)bh->b_data) + (offset & ROMBMASK), count);
        brelse(bh);
}
```

sys_open dependencia de las funciones

```
int sys_open(const char *filename, int flags, int mode) {

getname
        kmem_cache_alloc(names_cachep, SLAB_KERNEL)
        strncpy_from_user(tmp, filename, PATH_MAX + 1);


get_unused_fd
        find_next_zero_bit
        FD_SET

filp_open(tmp, flags, mode)
        open_namei
                mntget
                dget
                path_walk
                        cached_lookup
                        real_look
                                d_alloc
                                d_inode->i_op->lookup
                                        romfs_copyfrom
                                                bread
                                                memcpy
                                                brelse(bh);

        dentry_open
                get_empty_filp
fd_install(fd, f);                                      /*(4)*/

putname
        kmem_cache_free


     return fd;
}
```

getname →	kmem_cache_alloc stmcpy_from_user
get_unused →	find_next_zero_bit

<table>
<tr><td rowspan="3">filp_open →</td><td rowspan="2">open_namei →</td><td>mnget
dget</td><td></td></tr>
<tr><td>path_walk →</td><td>cached_lookup
real_lookup →</td></tr>
<tr><td>dentry_open →</td><td>get_empty_filp</td><td></td></tr>
<tr><td>fd_install</td><td></td><td></td><td></td></tr>
<tr><td>putname →</td><td>kmem_cache_free</td><td></td><td></td></tr>
</table>

<table>
<tr><td rowspan="2">romfs_lookup →</td><td>rmfs_copyfrom →</td><td>bread
memcpy →
brelse</td><td>→ …</td></tr>
<tr><td>iget
c_add</td><td></td><td></td></tr>
</table>

Principales estructuras de datos utilizadas

task_struct
files
file
filp
dentry
nameidata
inode