TEMA 1. VISIÓN GENERAL E INTRODUCCIÓN AL KERNEL

1.1. Introducción histórica a UNIX y Linux.

1.2. Visión general de UNIX.
- 1.2.1. Estructura del sistema.
- 1.2.2. Perspectiva del usuario.
  - 1.2.2.1. El sistema de archivos.
  - 1.2.2.2. El entorno de procesamiento.
  - 1.2.2.3. Primitivas de construcción de bloques.
- 1.2.3. Servicios del sistema operativo.
- 1.2.4. Aspectos del hardware.
  - 1.2.4.1. Interrupciones y excepciones.
  - 1.2.4.2. Niveles de ejecución del procesador.
  - 1.2.4.3. Manejo de memoria.

1.3. Introducción al *kernel* de UNIX
- 1.3.1. Arquitectura del sistema operativo.
- 1.3.2. Introducción a los conceptos del sistema.
  - 1.3.2.1. Subsistema de archivos.
  - 1.3.2.2. Subsistema de procesos.
    - 1.3.2.2.1. Contexto de un proceso.
    - 1.3.2.2.2. Estados de un proceso.
    - 1.3.2.2.3. Transiciones entre estados.
    - 1.3.2.2.4. Dormir y despertar.
- 1.3.3. Estructuras de datos del *kernel* (núcleo). Ventajas e inconvenientes.

1.4. Introducción a Linux.
- 1.4.1. Funciones del sistema operativo.
- 1.4.2. Descripción de Linux y de sus funcionalidades.
- 1.4.3. Estructura general del sistema operativo Linux.
- 1.4.4. Organización del código fuente del *kernel.*
- 1.4.5. Funcionamiento general del *kernel* de Linux.
  - 1.4.5.1. Implementación de una llamada al sistema.
  - 1.4.5.2. Creación de una llamada al sistema.
  - 1.4.5.3. Códigos de retorno.

1.1. INTRODUCCIÓN HISTÓRICA A UNIX Y LINUX.

1965: Laboratorios Bell, General Electric y MIT intentan desarrollar un nuevo Sistema Operativo: **MULTICS** (sistema operativo multiusuario interactivo). Objetivos del nuevo Sistema Operativo: (1) Dar servicio simultáneo a gran cantidad de usuarios. (2) Proporcionar gran capacidad de cálculo y almacenamiento. (3) Permitir a los usuarios compartir datos fácilmente.
1969: Primera Versión de MULTICS en Bell. No cumplió las expectativas y Laboratorios Bell abandona el proyecto ⇒ El proyecto muere.

Fin proyecto MULTICS: Algunos participantes intentan mejorar el entorno de programación desarrollado.

- Ken Thompson, Dennis Ritchie y otros diseñan un sistema de archivos → versión primitiva del sistema de archivos de UNIX.
- Thompson y Ritchie → Primera implementación de su diseño. (1) Versión primitiva del sistema de archivos UNIX. Lenguaje ensamblador. (2) Subsistema de procesos. (3) Conjunto de programas de utilidad. (4) Ha nacido **UNIX**.

Thompson, Ritchie y el lenguaje C ⇒ **1973**: UNIX se rescribe en C casi en su totalidad (90% aproximadamente), dejando sólo un 10% dependiente del lenguaje máquina de una computadora en concreto ⇒ Ya que C se caracteriza por su portabilidad, es decir, su independencia de la máquina, UNIX hereda esta característica convirtiéndose en uno de los sistemas operativos más portables.

AT&T: No puede comercializar productos informáticos ⇒ UNIX se extiende por universidades ⇒ más popularidad.

Popularidad micros ⇒ Desarrolladores lo mejoran ⇒ System V de AT&T y BSD de Berkeley.

UNIX es un potente sistema operativo que en su mayor parte es independiente de la máquina. Permite ejecutar programas, aporta un interfaz con un gran número de periféricos (impresoras, discos, cintas magnéticas, terminales, ...) para trabajar con ellos de forma cómoda, controlando el flujo de información entre la computadora central y dichos periféricos. Además posee un eficaz sistema de archivos, administrando el sistema de información a largo plazo. UNIX es un sistema multitarea, multiusuario e interactivo de propósito general.

La popularidad del S.O. UNIX es debida a: (1) Escrito en un lenguaje de alto nivel (C) ⇒ Fácil de leer, entender, modificar y mover a otras máquinas. (2) Interfaz de usuario simple pero completa. (3) Primitivas de construcción de programas complejos a partir de otros más simples.(4) Sistema de archivos jerárquico. (5) Fácil mantenimiento. (6) Implementación eficiente. (7) Formato consistente para los archivos ⇒ permite escribir programas con facilidad. (8) Interfaz consistente y simple con los dispositivos periféricos. (9) Hoy existen numerosas variantes comerciales (SCO, IBM, Digital, HP, SGI, Sun, etc.) que actúan en entornos cliente servidor, intranet, Internet, etc.

Por lo que respecta a Linux: **(1)** Linux aparece en 1991 como evolución de MINIX para el 80386. **(2)** En menos de un año, más de 100 programadores colaboran. **(3)** Forma de trabajo: Los fuentes (versiones) se difunden con la máxima frecuencia y cualquiera que quiera modificar o criticar algo, lo hace. Los más conocedores de esa área (los que la han programado) deciden si es útil y si lo es, lo incorporan. **(4)** Se portan las herramientas GNU de la FSF (gcc, gdb, bash, emacs, etc.) **(5)** En marzo de 1994 aparece Linux 1.0 en forma de "distribución". **(6)** En junio de 1996 se distribuye Linux 2.0, ya competitivo con otros UNIX. **(7)** Aparecen varias distribuciones (RedHat, Caldera, S.U.S.E, Slackware, Debian, etc., algunas con soporte oficial para empresas). **(8)** La FSF adopta Linux como Sistema Operativo Oficial: GNU/Linux. **(9)** Interés por parte de las empresas (Intel, Sun, Netscape, Lotus, Adobe, Corel, Oracle, Informix, Sysbase, etc.). **(10)** En enero de 1999 aparece Linux 2.2 con muchas mejoras y soporte para nuevos tipos de hardware. El primer número cambia cuando se da una evolución importante, el segundo es la versión y el tercero la revisión. Las versiones pares son estables y las impares inestables. **(11)** En enero de 2001 aparece Linux 2.4 con mejoras en el soporte a multiprocesadores, dispositivos como el USB, y acceso directo al HW gráfico (2D, 3D).

- Sistema interactivo. El usuario interactúa con el sistema en tiempo real sin necesidad de esperar colas de ejecución como ocurría en los sistemas batch.
- Sistema multiusuario. En un momento dado varios usuarios distintos pueden actuar en el mismo sistema, accediendo desde terminales distintas.
- Es un sistema multiprogramado (multitarea). Varios programas pueden estar simultáneamente en el sistema en diferentes áreas de memoria. Mientras que un trabajo está esperando operaciones de E/S otro puede estar usando la CPU. Si se pueden tener a la vez varios trabajos en memoria principal, la CPU puede mantenerse ocupada casi todo el tiempo.
- Es un sistema de tiempo compartido. El sistema UNIX divide el tiempo de la computadora en un número de partes, repartiéndolas entre los diferentes procesos. La computadora puede proporcionar servicio rápido e interactivo a varios usuarios e incluso procesar trabajos batch grandes en segundo plano cuando la CPU está ociosa.
- Es un sistema multiplataforma. UNIX es un sistema que proporciona compatibilidad para arquitecturas distintas.
- Oculta la arquitectura de la máquina, del usuario $\Rightarrow$ Fácil escribir programas que van a correr en hardware diversos.

1.2. VISIÓN GENERAL DE UNIX.

1.2.1. Estructura del Sistema.

UNIX es un sistema operativo de tiempo compartido (el sistema UNIX divide el tiempo de la computadora en un número de partes, repartiéndolas entre los diferentes procesos). El *kernel* del sistema es un programa que siempre está residente en memoria y, entre otros, brinda los siguientes servicios:

- Controla los recursos básicos.
- Controla los dispositivos periféricos (discos, terminales, impresoras, etc.).
- Permite a distintos usuarios compartir recursos y ejecutar sus programas.
- Proporciona un sistema de archivos que administra el almacenamiento de información (programas, datos, documentos, etc.).
- Es un sistema interactivo, permite el redireccionamiento de la E/S, tuberías y ejecución de procesos en background.

En un sentido más amplio, UNIX abarca también un conjunto de programas estándar, como pueden ser:

- Compilador de lenguaje C (cc).
- Editor de texto (vi).
- Intérprete de órdenes (sh, ksh, csh, ...).
- Programas de gestión de archivos y directorios (cp, rm, mv, mkdir, rmdir, ...).

Organización de un sistema UNIX cualquiera: Figura 1.1. El *hardware* suministra al sistema operativo servicios básicos.

En la Figura 1.1, se pueden ver distintos niveles dentro de la arquitectura UNIX. El nivel más externo (*Otros programas de aplicación*) no pertenece al sistema operativo, y si la máquina que esta implementada sobre el *hardware*, cuyos recursos queremos gestionar.

Directamente en contacto con el *hardware* está el segundo nivel, el *kernel* del sistema, este está escrito en C en su mayor parte aunque coexiste con código ensamblador. El *kernel* del sistema operativo es un programa que esta siempre residente en memoria y brinda servicios para controlar los recursos del sistema.

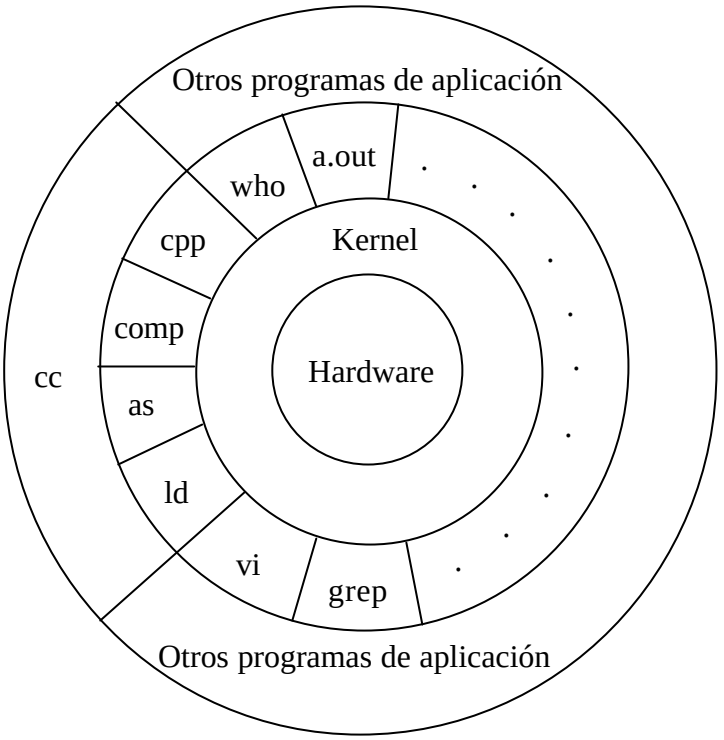

Figura 1.1. Organización de los sistemas UNIX.

Algunas funciones del sistema operativo se usan casi continuamente. Por ejemplo, la parte del sistema UNIX que se encarga de cambiar un programa por otro (tiempo compartido) se necesita muchas veces cada fracción de segundo. Todas aquellas funciones que se necesitan inmediatamente, se mantienen permanentemente en memoria. A la parte residente en memoria de un sistema operativo se llama *kernel*. El sistema UNIX ha dotado al *kernel* con relativamente pocas prestaciones, de forma que la mayoría de las funciones del sistema operativo las deben de proporcionar los programas de utilidad.

En el tercer nivel de la estructura que muestra la Figura 1.1 se encuentran los programas estándar de cualquier sistema UNIX (vi, grep, sh, who, ...) y programas generados por el usuario. Se debe destacar, que estos programas nunca van a actuar sobre el *hardware* de forma directa. Por ello, debe de existir algún mecanismo que permita indicarle al *kernel* que necesitamos operar sobre algún recurso *hardware*. Este mecanismo es lo que se conoce como llamadas al sistema (*system calls*). Así pues, cualquier programa que se esté ejecutando bajo el control de UNIX, cuando necesite hacer uso de alguno de los recursos que le brinda el sistema, deberá efectuar una llamada a alguna de las llamadas al sistema (*system calls*).

En un cuarto nivel tenemos aplicaciones que se sirven de otros programas ya creados para llevar a cabo su función. Estas aplicaciones, tampoco se comunican directamente con el *kernel*.

También hay que destacar que la jerarquía de programas no tiene por qué verse limitada a cuatro niveles. El usuario puede crear tantos niveles como necesite. Puede haber también programas que se apoyen en diferentes niveles y que se comuniquen con el *kernel* por un lado y con otros programas ya existentes por otro.

Sistema Operativo interactúa con el hardware.

- Da servicio a los programas.
- Aísla a los programas del *hardware*.
- Programas independientes del *hardware* $\Rightarrow$ *sencillo* moverlos entre sistemas UNIX sobre distintos *hardware*.

Sistema Operativo interacciona con procesos y usuarios.

- Interfaz sistema operativo – procesos de usuario.
  - **Llamadas al sistema**:
    + Solicitan al *kernel* que realice operaciones para el que invoca.
    + Pueden traer consigo intercambios de datos entre ambos.
- Interfaz sistema operativo – usuario.
  - **Comandos**: Ver Figura 1.1.

1.2.2. Perspectiva del Usuario

- Sistema de archivos.
- Entorno de procesamiento.
- Primitivas de construcción de bloques.

1.2.2.1. El Sistema de Archivos.

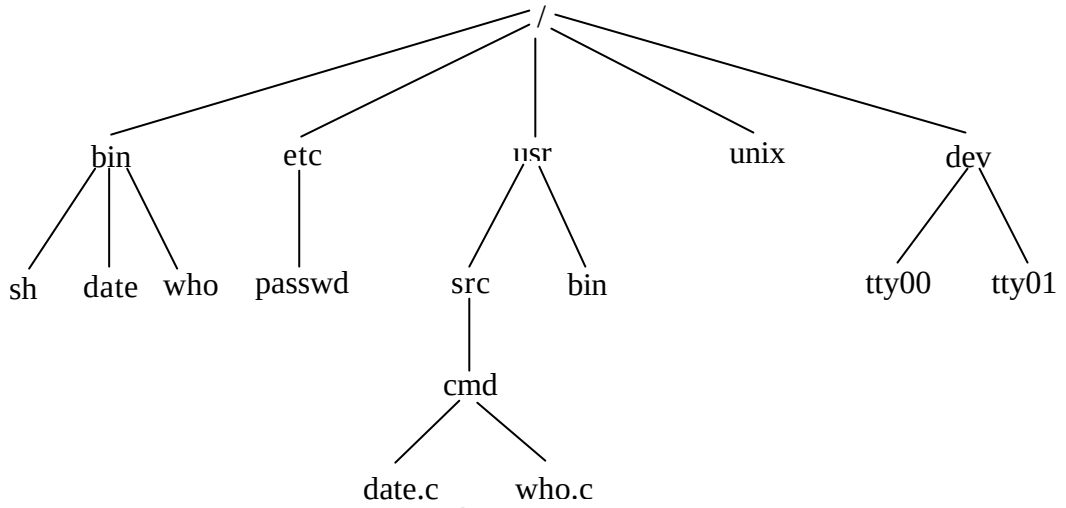

Figura 1.2. Ejemplo de árbol del sistema de archivos en UNIX.

Características:

- Estructura jerárquica: Sistema de archivos organizado en forma de árbol.
  - Nodo no-hoja es un archivo de directorio.
  - Nodos hoja → directorios, archivos regulares o de dispositivo.
- Tratamiento consistente de los datos de los archivos.
  - Programas UNIX.
    + No conocen el formato interno en que el *kernel* almacena los datos.
  - Tratan los datos como secuencia de bytes y las interpretan como quieran.
    + No existe relación de interpretación – formato almacenamiento.
  - Ejemplo: Directorios son como archivos regulares.
    + Sistema trata los datos del directorio como una secuencia de bytes.
    + Datos → Nombres de archivos en formato predefinido.
    + Programas (*ls*) interpretan la información correctamente.
- Posibilidad de crear y eliminar archivos.
- Crecimiento dinámico de los archivos.

- Protección de los datos de los archivos.
  - Permiso de acceso a los archivos → Controlado por un conjunto de permisos de acceso asociado al archivo.
    + Permisos de lectura, escritura y ejecución.
    + Para propietario, grupo y resto de usuarios del sistema.
    + Críticas al sistema de seguridad de acceso.
- Tratamiento de los dispositivos periféricos como archivos.
  - Dispositivos → Archivos especiales de dispositivo.
  - Ocupan lugar en la estructura de directorios como un archivo más.
  - Programas acceden a dispositivos igual que a un archivo regular.

1.2.2.2. El Entorno de Procesamiento.

- UNIX
  - Ejecución simultánea de varios procesos (multitarea o multiprocesamiento) sin un límite lógico.
    + Límites físicos: tamaño de la memoria, etc.
    + Degradación del sistema si el nivel de multiprocesamiento crece.
  - Varias instancias de un mismo programa → Código reentrante.
    + Llamadas al sistema para crear, terminar, sincronizar y especificar el comportamiento de procesos.
    + Permite ejecución síncrona y asíncrona.
    + Comentario: Hilos, Duplicación del código.

1.2.2.3. Primitivas de Construcción de Bloques.

- Filosofía UNIX → Proporcionar al usuario primitivas de construcción de programas.
  - Le permiten construir programas sencillos.
  - Utilización posterior como "bloques de construcción" para construir programas más complejos.
- *Primitiva 1*: Redireccionar la E/S.
  - Procesos → Acceso por defecto: archivo de entrada estándar, archivo de salida estándar y archivo de error estándar.
  - UNIX proporciona mecanismos para redireccionar cualquiera de ellos ⇒ Organización cuidadosa del sistema de archivos.
- *Primitiva 2*: Pipes o tuberías.
  - Mecanismo que permite el paso de datos entre un proceso escritor y un proceso lector sin necesidad de crear archivos temporales.
    + Escritor redirecciona la salida estándar hacia pipe.
    + Lector redirecciona su entrada estándar hacia el pipe.

1.2.3. Servicios del Sistema Operativo.

- El *kernel* está entre el *hardware* y los programas de usuario.
  - Sirve a programas de usuario → Llamadas al sistema (*system calls*).
  - Ver Figura 1.1.
- Servicios que ofrece el *kernel.*
  - Control de ejecución de procesos.
  - Planificación equitativa de procesos → Tiempo compartido.
  - Administración de la memoria principal → Procesos en ejecución.
    + Permite a procesos compartir parte del espacio de direcciones.
    + Protege el espacio de direccionamiento privado de procesos.
    + Gestiona la memoria libre en tiempos de escasez.
      * Dispositivo *swap* (intercambio[1]).
      * Sistema de intercambio o sistema paginado.

  - Administra la memoria secundaria eficientemente: SF.
  - Asigna la memoria secundaria para archivos de usuario.
    + Recupera el espacio de almacenamiento no utilizado.
    + Estructura el sistema de archivos de manera comprensible.
    + Protege los archivos de usuario ante accesos ilegales.
  - Permite a procesos el acceso controlado a dispositivos periféricos.
- El *kernel* proporciona estos servicios de una manera transparente.
- [1]El *intercambio* consiste en llevar los procesos cuyo tiempo de ocupación de memoria expira, a un área de memoria secundaria y traer de esa área de memoria secundaria los procesos a los que se les asigna tiempo de ocupación en memoria principal.

1.2.4. Aspectos del Hardware.

- Ejecución de un proceso en UNIX → dos niveles: *usuario* y *kernel.*
  - Proceso realiza llamada al sistema ⇒ modo de ejecución pasa de modo usuario a modo *kernel.*
    + Sistema operativo intenta servir una petición del usuario.
    + *Modo kernel*: Las llamadas al sistema (*system calls*) se ejecutan en modo *kernel* y para entrar en ese modo hay que ejecutar una sentencia de código máquina conocida como "trap". Es por esto, que las llamadas al sistema pueden ser invocadas directamente desde ensamblador y no desde C. En este modo están permitidas todas las instrucciones para el sistema operativo.
    + *Modo usuario*: en este modo no están permitidas instrucciones de E/S y de otros tipos para programas de usuario.
  - *Kernel* realiza tareas de gestión para procesos de usuario.
  - Diferencias entre los dos niveles son:
    + Acceso a instrucciones y datos.
    + Ejecución de instrucciones máquina privilegiadas.
- El *kernel* se ejecuta a petición de los procesos de usuario.
  - *Kernel* no son procesos ejecutándose en paralelo para servir a procesos de usuario.
  - Código ejecutado por el propio proceso en modo *kernel* o protegido.
    + Permite ejecución síncrona y asíncrona.
    + Organización alrededor de servidores.

1.2.4.1. Interrupciones y Excepciones.

- Las interrupciones son señales eléctricas manejadas directamente por un controlador de interrupciones (*hardware*) o codificando a nivel de software el índice para acceder a la tabla de descriptores de interrupciones (software).
- UNIX permite a dispositivos interrumpir la CPU asíncronamente.
  - Llegada de una interrupción.
  - El *kernel* salva contexto actual (imagen de lo que estaba haciendo el proceso).
  - Determina la causa de la interrupción.
    + Sirve la interrupción ⇒ Bloquea interrupciones menos prioritarias, sirve interrupciones de mayor prioridad.
  - Fin servicio de interrupción.
    + Restaura el contexto salvado.
    + Continúa como si no hubiera pasado nada.
- Distinción excepciones – interrupciones. Además de las interrupciones *hardware* y software, diversas condiciones de error de la CPU pueden causar la iniciación de una excepción. Las excepciones pueden servir: para estimular al sistema operativo para que suministre un servicio, para suministrar más memoria a un proceso, ...

1.2.4.2. Niveles de Ejecución del Procesador.

- El *kernel* debe prevenir la ocurrencia de interrupciones cuando se encuentra realizando operaciones críticas.
  - Peligro de corrupción de los datos.
  - Fija el nivel de ejecución del procesador ⇒ Enmascara interrupciones de ese y niveles inferiores, permite interrupciones de más alto nivel.

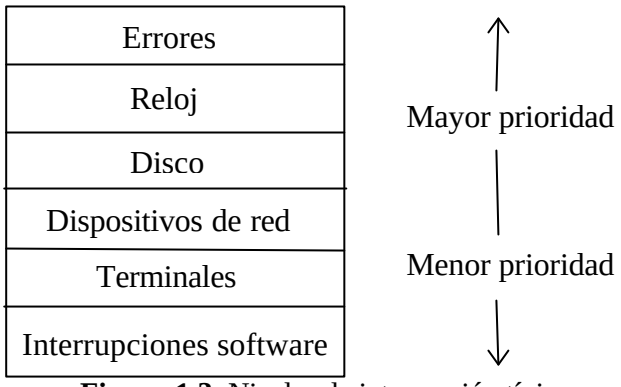

Figura 1.3. Niveles de interrupción típicos.

1.2.4.3. Manejo de Memoria.

- Compilación de un programa ⇒ el compilador genera direcciones para variables, funciones, etc. para una máquina virtual.
- El *kernel* reside permanentemente en memoria.
- Programa va a ejecutarse ⇒ *kernel* le asigna memoria principal.
  - Problema: Direcciones virtuales y físicas no coinciden.
  - *Kernel* debe coordinarse con el *hardware* ⇒ traducción direcciones virtuales a físicas.
    + Forma de realizar traducción depende del *hardware*.
    + Partes del sistema UNIX que tratan con ella serán dependientes de la máquina.
    + Micronúcleo.

1.3. INTRODUCCIÓN AL KERNEL DE UNIX.

1.3.1. Arquitectura del Sistema Operativo.

- UNIX ofrece al usuario dos entidades: *proceso* y *archivo*. Los dos conceptos centrales sobre los que se basa la arquitectura de UNIX son los procesos y los archivos. El *kernel* está pensado para facilitar servicios relacionados con el sistema de archivos y con el control de procesos.

Diagrama de bloques del *kernel*: Ver Figura 1.4.

- Módulos que componen el sistema operativo y relaciones entre ellos.
- Tres niveles: usuario, *kernel* y *hardware*.
  - Interfaz de llamadas al sistema y funciones de librería.
    + Frontera entre los programas de usuario y el *kernel.*
    + Llamadas al sistema: Forma de una llamada a una función C.
    + Librerías: Llamadas → primitivas.

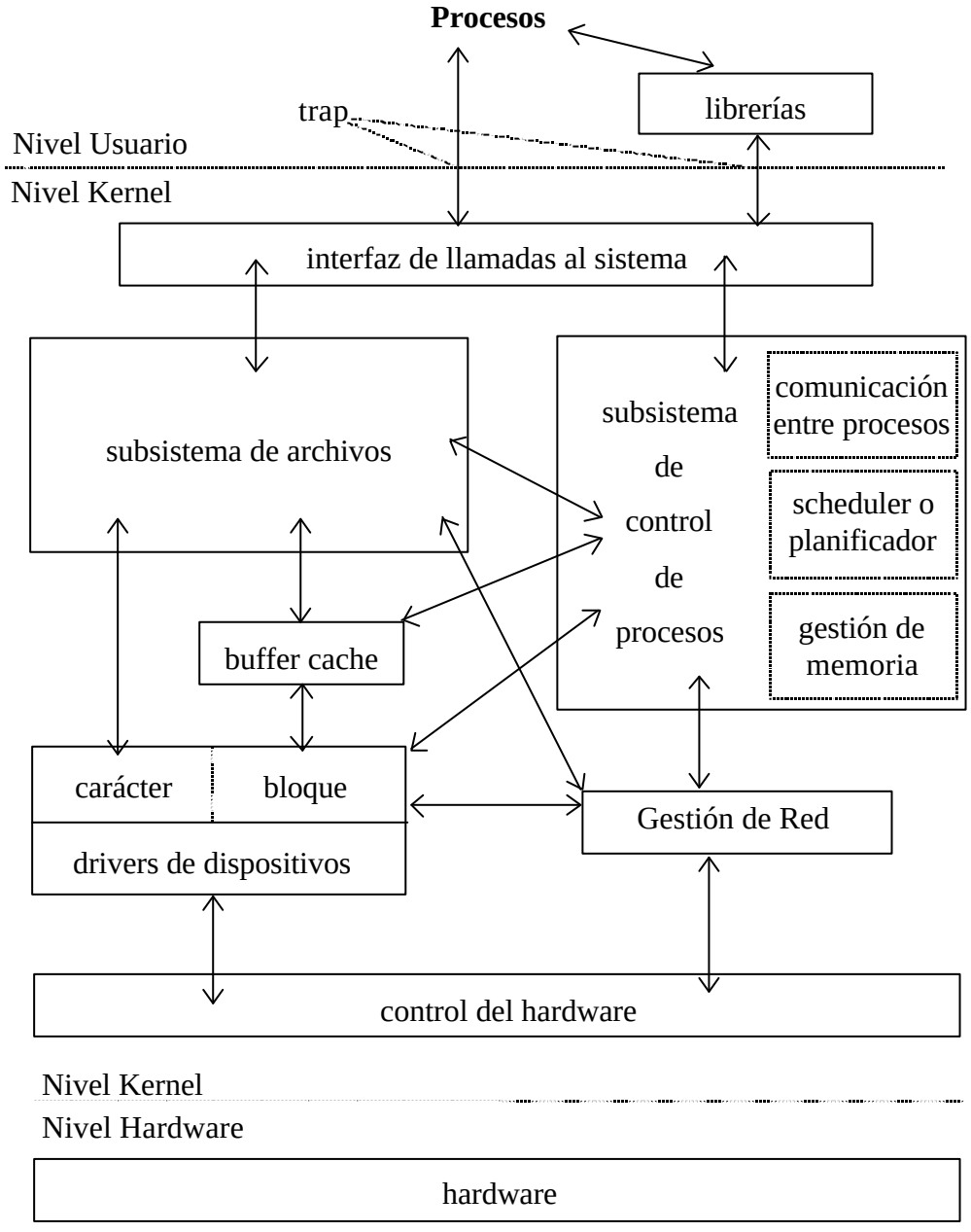

Figura 1.4. Diagrama de bloques del *kernel* del sistema.

La Figura 1.4. muestra los tres niveles que se van a estudiar en la arquitectura del sistema UNIX: *hardware*, *kernel* y usuario. Las llamadas al sistema y su librería asociada representan la frontera entre los programas de usuario y el *kernel*. La librería asociada a las llamadas del sistema es el mecanismo mediante el cual podemos invocar una llamada al sistema desde un programa C. Esta librería se linka (enlaza) por defecto al compilar cualquier programa C y se encuentra en el archivo */usr/lib/libc.a*. Los programas escritos en lenguaje ensamblador pueden invocar directamente a las llamadas al sistema sin necesidad de ninguna librería intermedia.

Las llamadas al sistema se ejecutan en modo *kernel* y para entrar en este modo hay que ejecutar una sentencia en código máquina conocida como *trap* (o interrupción software). Es por esto que las llamadas al sistema pueden ser invocadas directamente desde ensamblador y no desde C.

Llamadas al sistema. Un proceso que se ejecuta en modo usuario no puede acceder directamente a los recursos de la máquina, para ellos debe ejecutar llamadas al sistema. *Concepto*: Una llamada al sistema es una petición transmitida por un proceso al *kernel*. Este último trata la petición en modo *kernel*, con todos los privilegios, y envía el resultado al proceso, que prosigue su ejecución. Bajo UNIX, la llamada al sistema provoca un cambio: el proceso ejecuta una instrucción del procesador que le hace pasar a modo *kernel*, seguidamente ejecuta una función de tratamiento vinculada a la llamada al sistema que ha efectuado, y luego vuelve al modo usuario para proseguir su ejecución. De este modo, el propio proceso trata su llamada al sistema, ejecutando una función del *kernel*. Esta función se supone que es fiable y puede ejecutarse en modo privilegiado, contrariamente al proceso ejecutado en modo usuario.

Subsistemas que forman la arquitectura UNIX. Subsistema de archivos y subsistema de control de procesos.

- Subsistema de archivos → Gestión de archivos.
  - Asigna espacio, administra espacio libre, controla acceso a archivos y devuelve datos.
  - Accede a datos de archivos → mecanismo de buffering.
  - Interacción programas - subsistema archivos: llamadas al sistema específicas.

El subsistema de archivos controla los recursos del sistema de archivos y tiene funciones como: reservar espacio para los archivos, administrar el espacio libre, controlar el acceso a los archivos, permitir el intercambio de datos entre los archivos y el usuario, etc. Los procesos interaccionan con el subsistema de archivos a través de unas llamadas específicas (open, close, read, write, status, chown, etc.).

El subsistema de archivos se comunica con los dispositivos de almacenamiento secundario (discos duros, unidades de cinta, etc.) a través de manejadores de dispositivos (*device drivers*). Los manejadores de dispositivos se encargan de proporcionar el protocolo de comunicación (*handshake*) entre el *kernel* y los periféricos. Se consideran dos tipos de dispositivos según la forma de acceso: dispositivos modo bloque (*block devices*) y dispositivos modo carácter (*row devices*). El acceso a los dispositivos en modo bloque se lleva a cabo con la intervención de *buffers* que mejoran enormemente la velocidad de transferencia. El acceso a dispositivos en modo carácter se lleva a cabo de forma directa, sin la intervención de *buffers*. Un mismo dispositivo físico puede ser manejado tanto en modo bloque como en modo carácter, dependiendo de qué manejador (driver) utilicemos para acceder a él.

- Subsistema de control de procesos → Gestión de procesos.
  - Planificación, sincronización y comunicación de procesos y gestión de memoria.
  - Interacción con subsistema de archivos → cuando se carga un archivo en memoria para su ejecución.
  - Llamadas al sistema de control de procesos: *fork*, *exec*, *exit*, *wait*, etc.
  - Módulo gestión de memoria → Controla asignación memoria.
    - Políticas gestión memoria: swapping y demanda de páginas (paginación).
  - Módulo planificador (*scheduler*) → Asigna la CPU a los procesos.
- Control del *hardware* → Responsable del manejo de interrupciones y de la comunicación con la máquina.

El subsistema de control de procesos es el responsable de la planificación de los procesos (*scheduler*), su sincronización, comunicación entre los mismos (*IPC – Inter Process Comunication*) y del control de la memoria principal. Algunas de las llamadas utilizadas para controlar procesos son: *fork*, *exec*, *exit*, *wait*, *brk*, *signal*, etc.

El módulo de gestión de memoria se encarga de controlar qué procesos están cargados en memoria principal en cada instante. Si en un momento determinado no hay suficiente memoria principal para todos los procesos que lo solicitan, el gestor de memoria debe recurrir a mecanismos de intercambio (*swapping*) para que todos los procesos tenga derecho a un tiempo mínimo de ocupación de la memoria y se puedan ejecutar.

El intercambio (*swapping*) consiste en llevar los procesos cuyo tiempo de ocupación de la memoria expira, a una memoria secundaria (que normalmente es el área que se dedica al intercambio en el disco (área de *swap*) y que se monta como un sistema de archivos aparte) y traer de esa memoria secundaria los procesos a los que se les asigna tiempo de ocupación de la memoria principal. Al módulo gestor de los mecanismos de intercambio se le conoce también como intercambiador (*swapper*).

El planificador o *scheduler* se encarga de gestionar el tiempo de CPU que tiene asignado cada proceso. El *scheduler* entra en ejecución cada *cuantum* de tiempo y decide si el proceso actual tiene derecho a seguir ejecutándose (esto depende de su prioridad y de sus privilegios) o ha de conmutarse de contexto (asignarle la CPU a otro proceso).

La comunicación entre procesos puede realizarse de forma asíncrona (señales) o síncrona (colas de mensajes, semáforos).

Por último, el módulo de *control del hardware* es la parte del *kernel* encargada del manejo de las interrupciones y de la comunicación con la máquina. Los dispositivos pueden interrumpir a la CPU mientras está ejecutando un proceso. Si esto ocurre, el *kernel* debe reanudar la ejecución del proceso después de atender a la interrupción. Las interrupciones no son atendidas por procesos, sino por funciones especiales, codificadas en el *kernel*, que son invocadas durante la ejecución de cualquier proceso.

1.3.2. Introducción a los Conceptos del Sistema.

1.3.2.1. Subsistema de Archivos.

Un sistema de archivos permite realizar una abstracción de los dispositivos físicos de almacenamiento de la información para que sean tratados a nivel lógico, como una estructura de más alto nivel y más sencilla que la estructura de su arquitectura *hardware* particular.

Características del sistema de archivos. El sistema de archivos de UNIX se caracteriza por:

- Poseer una estructura jerárquica.
- Realizar un tratamiento consistente de los datos de los archivos.
- Poder crear y borrar archivos.
- Permitir un crecimiento dinámico de los archivos.
- Proteger los datos de los archivos.
- Tratar los dispositivos y periféricos (terminales, unidades de cinta, etc.) como si fuesen archivos.

Organización del sistema de archivos.

- Los sistemas de archivos suelen estar situados en dispositivos de almacenamiento modo bloque, tales como cintas o discos.
- Un sistema UNIX puede manejar una o varias unidades de disco físicas → cada una puede contener uno o varios sistemas de archivos. Los sistemas de archivos son particiones lógicas del disco.
- El *kernel* trata a nivel lógico con los sistemas de archivos y no trata directamente con los discos a nivel físico.
  - Sistema de archivos → dispositivo lógico. Es decir, cada disco es considerado como un dispositivo lógico que tiene asociados unos números de dispositivo (*minor number* y *major number*). Estos números se utilizan para indexar dentro de una tabla de funciones, cuál tenemos que utilizar para acceder al controlador del disco.
  - Driver de disco: direcciones de dispositivo lógico → direcciones de dispositivo físico. Es decir, el controlador del disco se va a encargar de transformar las direcciones lógicas de nuestros sistema de archivos a direcciones físicas del disco.
- Sistema de archivos.
  - Secuencia de bloques lógicos de tamaño homogéneo dentro de un mismo sistema de archivos. El tamaño de cada bloque es el mismo para todo el sistema de archivos y suele ser múltiplo de 512 bytes.
  - A pesar de que el tamaño de un bloque es homogéneo en un sistema de archivos, puede variar de un sistema a otro de una misma configuración UNIX con varios sistema de archivos. El tamaño elegido para un bloque va a influir en las prestaciones globales del sistema. Por un lado interesa que los bloques sean grandes para que la velocidad de transferencia entre el disco y memoria sea grande. Sin embargo, si los bloques lógicos son demasiado grandes, la capacidad de almacenamiento del disco se puede ver desaprovechada cuando abundan los archivos pequeños que no llegan a ocupar un bloque completo (*fragmentación*). Valores típicos en bytes para el tamaño de un bloque son: 512, 1024 y 2048.

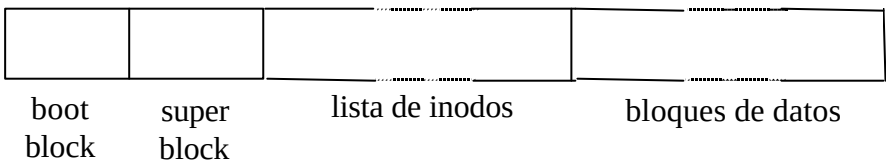

Figura 1.6. Primer nivel en la estructura del sistema de archivos en el UNIX System V.

En la Figura 1.6 podemos ver cuatro partes que forman la estructura del sistema de archivos:

- Un sistema de archivos tiene la siguiente estructura:
  - *Boot block* (bloque de arranque).
    - Ocupa la parte del principio del sistema de archivos, típicamente el primer sector, y puede contener el código de arranque.
    - Este código de arranque es un pequeño programa que se encarga de buscar el sistema operativo y cargarlo en memoria para inicializarlo.
    - Uno en cada sistema de archivos → Vacíos, en los sistemas de archivos que no son de arranque.
  - *Superblock*.
    - Describe el estado del sistema de archivos. Contiene información acerca de su tamaño, número total de archivos que puede contener, qué espacio libre queda, etc.

  - *Lista de nodos índice* (inodos).
    + Se encuentra a continuación del *superbloque*.
    + Durante el proceso de arranque del sistema, el *kernel* lee la lista de inodos del disco y carga una copia en memoria, conocida como tabla de inodos. Las operaciones que haga el subsistema de archivos (parte del código del *kernel*) van a involucrar a la tabla de inodos pero no a la lista de inodos → mayor velocidad en el acceso a disco debido a que esta tabla se encuentra en memoria.
    + Esta lista tiene una entrada por cada archivo donde se guarda una descripción del mismo: situación del archivo en el disco, propietario, permisos de acceso, fecha de actualización, etc.
    + Tamaño configurable. Es decir, el administrador del sistema es el encargado de especificar el tamaño de la lista de nodos índice al configurar el sistema.
    + Inodo fundamental → inodo raíz del sistema de archivos.
  - *Bloques de datos*.
    + Empieza a continuación de la lista de nodos índice y ocupa el resto del sistema de archivos.
    + En esta zona es donde se encuentran los datos de archivos y datos administrativos a los que hace referencia la lista de i-nodos.
    + Cada uno de los bloques destinados a datos sólo puede ser asignado a un archivo, tanto si lo ocupa totalmente como sino.

Representación interna y gestión de archivos.

- Estructuras de datos: Tabla de inodos, tabla de archivos y tabla de descriptores de archivos de usuario.
- Representación → inodo. Cada archivo en UNIX tiene asociado un inodo.
  - Distribución de datos del archivo en el disco. Archivo ↔ asociado a un inodo.
  - Campos que componen un inodo: Identificador del propietario del archivo, tipo del archivo, tipo de acceso al archivo, tiempos de acceso al archivo, número de enlaces del archivo, entradas para los bloques de dirección de datos y tamaño del archivo.
- Gestión de archivos → Además de la tabla de inodos, el *kernel* mantiene en memoria otras dos tablas que contienen información necesaria para poder manipular un archivo: *tabla de archivos* y *tabla de descriptores de archivos de usuario*.
  - *Tabla de inodos* es una copia en memoria de la lista de inodos que hay en el disco, a la que se le añade información adicional. (estado del inodo, número de dispositivo lógico, número del inodo, ...)
  - *Tabla de archivos* es una estructura global del *kernel* y en ella hay una entrada por cada archivo distinto que los procesos del *kernel* o los procesos del usuario tienen abiertos. La tabla de archivos es una estructura de datos orientadas a objetos. Cada vez que un proceso abre o crea un archivo nuevo, se reserva una nueva entrada en la tabla. Cada entrada de la tabla contiene un bloque de datos y un array de punteros a funciones que traducen las operaciones genéricas que se pueden realizar con los archivos (leer, escribir, cerrar, posicionar, ...) en acciones concretas asociadas a cada tipo de archivo. Además, cada entrada de la tabla de archivos tiene también un puntero a una estructura de datos que contiene información del estado actual del archivo. (inodo asociado, permisos de acceso para el proceso, ...)
  - *Tabla de descriptores de archivos de usuario* es una estructura local a cada proceso. Esta tabla identifica todos los archivos abiertos por un proceso. Cuando utilizamos las llamadas *open*, *creat*, *dup* o *link*, el *kernel* devuelve un descriptor de archivo, que es un índice para poder acceder a las entradas de la tabla anterior (tabla de archivos). En cada una de las entradas de la tabla hay un puntero a una entrada de la tabla de archivos del sistema.

- Gestión de las tablas. Los procesos no van a manipular directamente ninguna de las tablas anteriores (esto lo hace el *kernel*), si no que van a acceder a los archivos manipulando su descriptor asociado, que es un número. Cuando un proceso utiliza una llamada para realizar una operación sobre un archivo, le va a pasar al *kernel* el descriptor de ese archivo. El *kernel* va a utilizar este número para acceder a la tabla de descriptores de archivos del proceso y buscar en ella cuál es la entrada de la tabla de archivos que le da acceso a su inodo. Este mecanismo puede parecer artificioso, pero ofrece una gran flexibilidad cuando queremos que un proceso acceda simultáneamente a un mismo archivo en modos distintos, o que varios procesos compartan archivos.

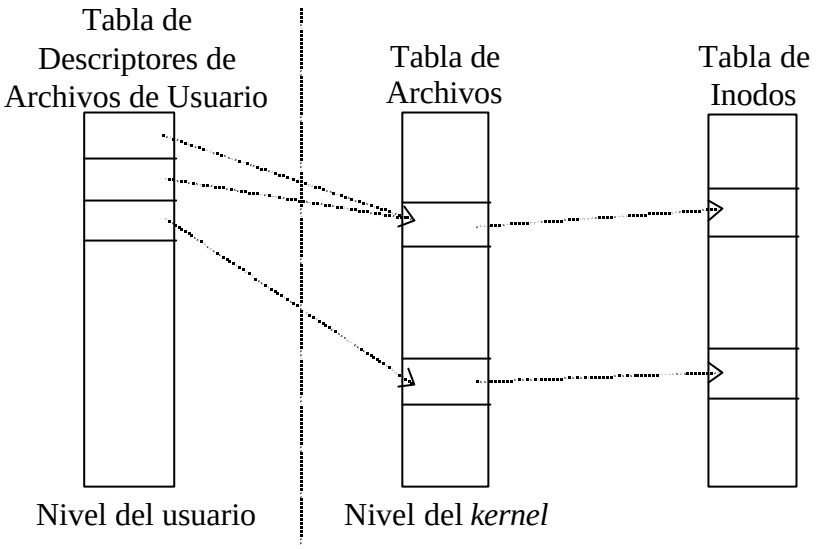

Figura 1.5. Descriptores de archivos, Tabla de Archivos y Tabla de inodos.

1.3.2.2. Subsistema de Procesos.

Programa.

- Un programa es una colección de instrucciones y de datos que se encuentran almacenados en un archivo ordinario (tipo del archivo). Este archivo tiene en su inodo un atributo (tipo de acceso al archivo) que lo identifica como ejecutable. Puede ser ejecutado por el propietario, por el grupo y por el resto de los usuarios, dependiendo de los permisos que tenga el archivo.

Proceso.

- Cuando un programa es leído del disco (a través del sistema de archivos) por el *kernel* y es cargado en memoria para ejecutarse, se convierte en un *proceso*.
- Ejecución de programa $\Rightarrow$ Conjunto de bytes que la CPU interpreta como instrucciones máquina (texto), datos y la pila.
- En un proceso no sólo hay una copia del programa, sino que además el *kernel* le añade información adicional para poder gestionarlo.
- Se comunican con otros procesos y con el resto del sistema vía llamadas al sistema.

Creación de proceso.

- Sistema UNIX → Un proceso es la entidad creada como resultado de *fork*.
  - Todos los procesos (creados mediante una llamada a *fork*) excepto el proceso 0 → creado a mano al arrancar el sistema.
    + El proceso 0 es especial (PID 0), éste es creado cuando arranca el sistema, y después de hacer una llamada *fork* se convierte en el proceso → *swapper* (encargado de la gestión de la memoria virtual).
    + El proceso hijo creado es el proceso con *PID* 1 → *init*, que es el ancestro de todos los demás procesos del sistema y es el encargado de arrancar los demás procesos del sistema según la configuración que se indica en el archivo */etc/inittab*.
  - Relación familiar: padres e hijos. El proceso que llama a *fork* se conoce como padre y el proceso creado es el proceso hijo. Todos los procesos tienen un único proceso padre, pero pueden tener varios procesos hijos.

Identificación de un proceso.

- El *kernel* identifica a cada proceso por su *PID* (*process identification*), que es un número asociado a cada proceso y que no cambia durante el tiempo de vida de éste.

Ejecución de un proceso.

- Las partes del programa ejecutable son:
  - Un conjunto de cabeceras que describen los atributos del archivo.
  - Un bloque donde se encuentran las instrucciones en lenguaje máquina del programa. Este bloque se conoce en UNIX como texto del programa.
  - Un bloque dedicado a la representación en lenguaje máquina de los datos que deben ser inicializados cuando arranca la ejecución del programa.
    + Valores iniciales para cuando comience su ejecución.
    + Aquí también está incluida una indicación de cuánto espacio de memoria debe reservar el *kernel* para estos datos.
    + Indicación del espacio que el *kernel* tiene que asignar para datos no inicializados, denominado *bss* (*block started symbol*). El *kernel* inicializa, en tiempo de ejecución, esta zona a valor 0.
  - Otras secciones, como información de la tabla de símbolos.
- Carga del programa ejecutable en memoria.
  - Llamada al sistema *exec*.
  - Proceso → formado por tres regiones o bloques fundamentales que se conocen como segmentos.
    + El segmento de texto. Contiene las instrucciones que entiende la CPU de nuestra máquina. Este bloque es una copia del bloque de texto del programa.
    + El segmento de datos. Contiene los datos que deben ser inicializados al arrancar el proceso. Si el programa ha sido generado por un compilador de C, en este bloque estarán las variables globales y las estáticas. Se corresponde con el bloque *bss* del programa.
    + El segmento de pila. Lo crea el *kernel* al arrancar el proceso y su tamaño es gestionado dinámicamente por el *kernel*. La pila se compone de una serie de bloques lógicos, llamados marcos (*frames*) de pila, que son introducidos cuando se llama a una función y son sacados cuando se vuelve de la función. Un marco (*frame*) de pila se compone de los parámetros de función, las variables locales de la función y la información necesaria para restaurar el marco (*frame*) de pila anterior a la llamada a la función (dentro de esta información se incluyen el contador de programa y el puntero de pila anterior a la llamada a la función). En los programas fuente no se incluye código para gestionar la pila (a menos que estén escritos en ensamblador), es el compilador quien incluye el código necesario para controlarlo.

Dos modos de ejecución para los procesos → el sistema maneja sendas pilas por separado. Pila del modo usuario y la pila del modo *kernel*.

* Pila del modo usuario. Contiene los argumentos, variables locales y otros datos relativos a funciones que se ejecutan en modo usuario.
* Pila del modo *kernel*. Contiene los marcos (*frames*) de pila de las funciones que se ejecutan en modo *kernel* (estas funciones son las llamadas al sistema).

– UNIX es un sistema que permite multiproceso (ejecución de varios procesos simultáneamente). El *scheduler* es la parte del *kernel* encargar de gestionar la CPU y determinar qué proceso pasa a ocupar tiempo de CPU en un determinado instante.
– Un mismo programa puede estar siendo ejecutado en un instante determinado por varios procesos a la vez.

Estructuras de datos.

Todo proceso tiene asociada una entrada en la *Tabla de Procesos* y un *Área de Usuario* (*U-area*). Estas dos estructuras van a describir el estado del proceso y le van a permitir al *kernel* su control.

- La *Tabla de Procesos* tiene campos que van a ser accesibles desde el *kernel*, pero los campos del *Área de Usuario* sólo necesitan ser visibles por el proceso. Una posición (entrada) por proceso.
  – Algunos de los campos que tiene cada una de las entradas de la *Tabla de Procesos* son: campo de estado, campo para localizar el proceso y su área de usuario, identificadores de usuario (*UID*), identificadores de proceso (*PID*), descriptores de eventos, parámetros de planificación, campo para señales, temporizadores, ...
- El *Área de Usuario* (*U-area*). Contiene datos privados manipulables por el *kernel* que son necesarios sólo cuando el proceso se está ejecutando.
  – Algunos de los campos de que consta esta *Área de Usuario* (*U-area*) son los siguientes: puntero a la entrada de la *Tabla de Procesos* correspondiente al proceso que pertenece el *Área de Usuario*, identificadores del usuario real y efectivo (*UID*), terminal de inicio, registro de errores, campo de valor de retorno, parámetros de entrada/salida, directorio de trabajo actual y directorio raíz, tabla de descriptores de archivo, campos de límite, máscara de permisos, temporizadores, ...
- Tabla de regiones por proceso. Las regiones son zonas lógicas que el sistema operativo divide el espacio de direcciones virtuales del proceso (área contigua del espacio de direcciones virtual).
- Tabla de regiones → Entradas describen atributos de la región (texto/datos, privada/compartible, etc).

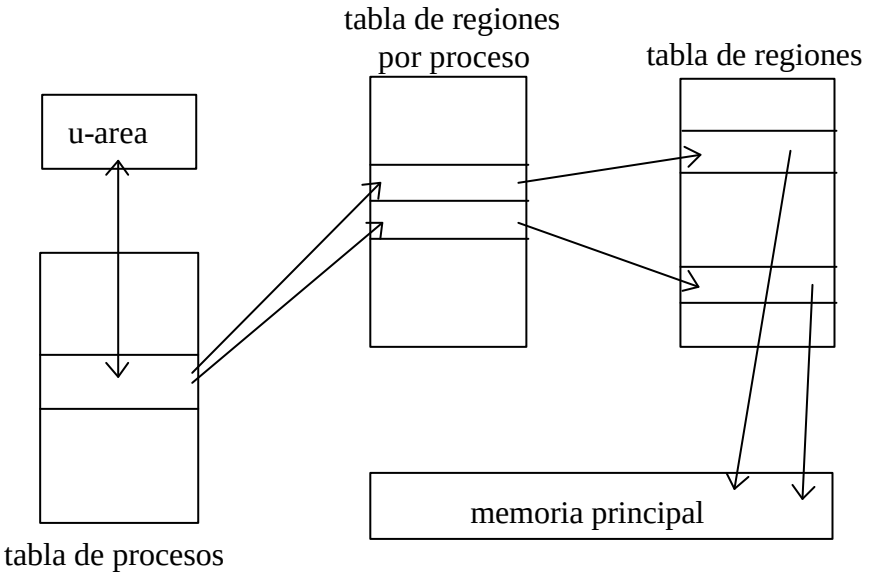

Figura 1.7. Estructuras de datos para los procesos.

Modo de operación.

- Ejecución de programas desde un proceso. Proceso realiza una llamada a *exec*. Básicamente, el resultado que se consigue con estas funciones (familia de funciones *exec*) es cargar un programa en la zona de memoria del proceso que ejecuta la llamada, sobrescribiendo los segmentos del programa antiguo con los del nuevo. El contexto del nivel de usuario del proceso que llama a *exec* deja de ser accesible y es reemplazado de acuerdo con el nuevo programa. Desde un programa se llama a otro programa.
  - El *kernel* asigna regiones para texto, datos y pila.
  - Libera las regiones del proceso que realiza la llamada.
- Creación de procesos. La única forma de crear un proceso en UNIX es mediante la llamada a *fork*. Proceso realiza una llamada a *fork*. El proceso que invoca a *fork* se llama proceso padre y el proceso creado es el proceso hijo.
  - El *kernel* duplica espacio de direccionamiento.
  - Padre e hijo comparten regiones de memoria si es posible.
  - Realiza copias físicas en caso contrario. Es decir, a la salida de *fork*, los dos procesos tienen una copia idéntica del contexto del nivel de usuario excepto el valor de *PID*, que para el proceso padre toma el valor del *PID* del proceso hijo y para el proceso hijo toma el valor 0. El proceso 0, creado por el *kernel* cuando arranca el sistema, es el único que no se crea vía una llamada *fork*.
- Proceso llama a *exit*. Esta llamada termina la ejecución de un proceso y le devuelve el valor de *status* al sistema.
  - El *kernel* libera las regiones que estaba usando. El contexto del proceso es descargado de memoria, lo que implica que la tabla de descriptores de archivos es cerrada y sus archivos asociados cerrados, si no quedan más procesos que los tengan abiertos.

1.3.2.2.1. Contexto de un Proceso.

- El contexto de un proceso es su estado, definido por:
  - Su código.
  - Los valores de sus variables de usuario globales y de sus estructuras de datos.
  - El valor de los registros de la CPU.
  - Los valores almacenados en su entrada de la tabla de procesos y en su área de usuario.
  - Y el contenido de sus pilas (*stacks*) de usuario y *kernel.*
- El código del sistema operativo y sus estructuras de datos globales, son compartidas por todos los procesos, pero no son consideradas como parte del contexto del proceso.
- Cuando se ejecuta un proceso, se dice que el sistema se está ejecutando en el contexto de un proceso. Cuando el *kernel* decide que debe ejecutar otro proceso, realiza un *cambio de contexto*.
  - Este cambio de contexto está permitido bajo ciertas condiciones.
  - Este cambio de contexto da lugar a que el *kernel* guarde la información necesaria para poder continuar con la ejecución del proceso interrumpido, más tarde, en el mismo punto donde lo dejó. De igual manera, cuando el proceso cambia de *modo usuario* a *modo kernel*, el *kernel* guarda información para cuando el proceso tenga que volver a modo usuario. Sin embargo, el cambio de modo usuario a modo *kernel* y viceversa, no se contempla como un cambio de contexto.
- Desde el punto de vista formal, el contexto de un proceso es la unión de su contexto del nivel de usuario, su contexto de registro y su contexto del nivel de sistema.
  - El *contexto de nivel de usuario* se compone de:
    + Los segmentos de texto, datos y pila del proceso.
    + Las zonas de memoria compartida que se encuentran en la zona de direcciones virtuales del proceso.
    + Las partes del espacio de direcciones virtuales que periódicamente no residen en memoria principal debido al *swapping* o a la paginación.

  - El *contexto de registros* se compone de las siguientes partes:
    + El contador de programa.
    + El registro de estado del procesador (*PS*).
    + El puntero de la pila.
    + Registros de propósito general.
  - El *contexto del nivel de sistema* de un proceso tiene una parte estática y otra parte dinámica. Todo proceso tiene una única parte estática del contexto de nivel de usuario, pero puede tener un número variable de partes dinámicas.
    + La entrada en la tabla de procesos.
    + El área de usuario.
    + Entradas de la tabla de regiones por proceso, tabla de regiones y tabla de páginas.
    + La pila de *kernel.*
    + Serie de capas que se almacenan en modo de pila y cada capa contiene la información necesaria para poder recuperar la capa anterior, incluyendo el contexto de registro de la capa anterior.
- El *kernel* introduce una capa de contexto cuando se produce una interrupción, una llamada al sistema o un cambio de contexto. Las capas de contexto son extraídas de la pila cuando el *kernel* vuelve del tratamiento de una interrupción, el proceso vuelve al modo usuario después de ejecutar una llamada al sistema o cuando se produce un cambio de contexto. La capa introducida es la del último proceso que se estaba ejecutando y la extraída es la del proceso que se va a pasar a ejecutar. Cada una de las entradas de la tabla de procesos contiene información suficiente para poder recuperar las capas de contexto.

1.3.2.2.2. Estados de un Proceso.

- Los estados de un proceso describen el tiempo de vida de un proceso. En este conjunto de estados, cada uno tiene una características determinadas.
- De forma simple, los estados por los que puede atravesar un proceso son los siguientes:
  - (1) El proceso se está ejecutando en modo usuario.
  - (2) El proceso se está ejecutando en modo *kernel.*
  - (3) El proceso no se está ejecutando, pero está listo para hacerlo tan pronto como el *scheduler* lo escoja. Puede haber varios procesos simultáneamente en este estado.
  - (4) El proceso está durmiendo. Un proceso entra en este estado cuando no puede proseguir su ejecución porque está esperando a que se complete una operación de E/S.
- En un sistema monoprocesador no puede haber más de un proceso ejecutándose a la vez. Así, en los dos modos de ejecución (usuario y *kernel*: estados (1) y (2)) sólo podrá haber un proceso en cada estado.

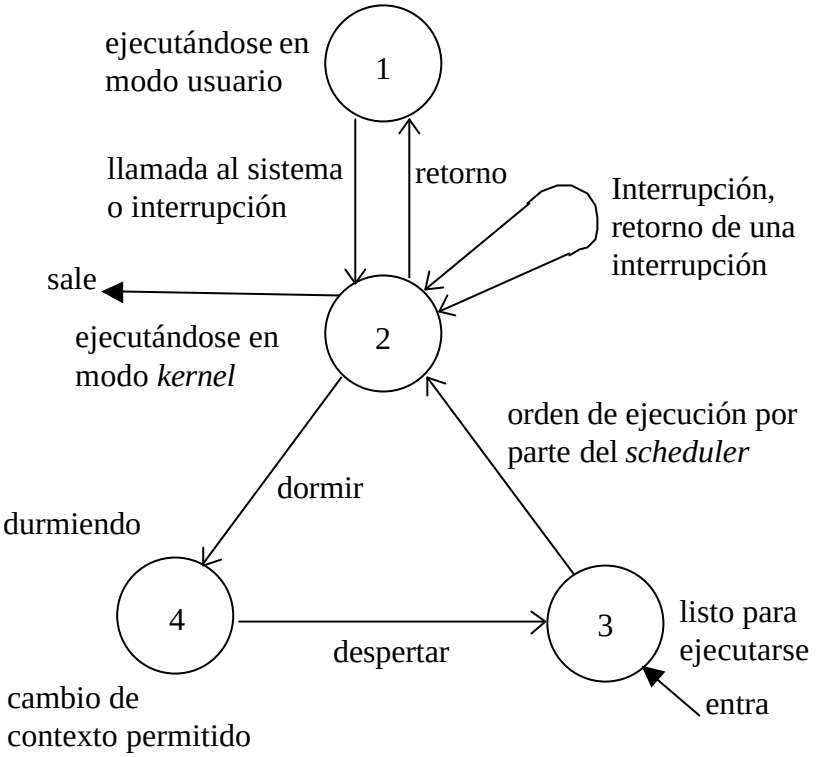

Figura 1.8. Estados de un proceso y transiciones.

1.3.2.2.3. Transiciones entre Estados.

- Procesos se mueven entre estados siguiendo reglas bien definidas. Es decir, que un proceso no permanece siempre en un mismo estado, sino que está continuamente cambiando de acuerdo con unas transiciones bien establecidas. Estos cambios de estados vienen impuestos por la competencia que existe entre los procesos para compartir un recurso tan escaso como es la CPU (*scheduler*).
- Diagrama de transición de estados como el de la Figura 1.8. Un diagrama de transición de estados es un grafo dirigido, cuyos nodos representan los estados que pueden alcanzar los procesos y cuyos arcos representan los eventos que hacen que un proceso cambie de un estado a otro.
- Varios procesos ejecutándose a la vez → *tiempo compartido*.
  - Todos los procesos podrían estar en modo *kernel*.
  - Problemas de corrupción de estructuras de datos globales del *kernel*.
  - El *kernel* mantiene la consistencia del sistema, prohibiendo cambios de contexto arbitrarios y controlando la ocurrencia de interrupciones.
    + El *kernel* permite el cambio de contexto cuando un proceso ha estado ejecutándose en modo *kernel* → durmiendo en memoria.
    + El *kernel* no es requisable → Consistencia de estructuras de datos.
    + *Kernel* entra en zonas críticas de código → aumenta nivel de ejecución del procesador para prevenir interrupciones.
    + El *scheduler* expulsa procesos que se están ejecutando en modo usuario de manera que no monopolicen la CPU.

1.3.2.2.4. Dormir y Despertar.

- Procesos duermen en un evento.
  - Esperan que ocurra otro evento ⇒ Despiertan ⇒ Pasan a listo para ejecutarse.
- Puede haber varios procesos dormidos en un evento ⇒ se despertarán todos ellos.
- Procesos dormidos no consumen recursos de la CPU.
  - El *kernel* no chequea constantemente si un proceso está todavía durmiendo.
  - Espera la ocurrencia del evento (despertar) y despierta al proceso.

1.3.3. Estructuras de Datos del Kernel. Ventajas y Desventajas.

- Estructuras de datos del *kernel* ocupan tablas de tamaño fijo, en lugar de espacio asignado dinámicamente.
  - Ventaja de está implementación es la sencillez del código del *kernel.*
  - Inconveniente: limita el número de entradas en cada estructura de datos al valor que se haya dado en la configuración del sistema.

- Estructuras de datos del *kernel*, estructuras dinámicas de tamaño variable.
  - Ventaja de está implementación es la posibilidad de manipular y gestionar estructuras de datos dinámicas que se adapten a la carga del sistema en tiempo de ejecución.
  - Inconveniente: código más complicado de desarrollar para que sea lo más eficiente posible.

1.4. INTRODUCCIÓN A LINUX.

1.4.1. Funciones del Sistema Operativo.

El sistema operativo UNIX se ha desarrollado en un lenguaje de alto nivel (lenguaje C), a diferencia de otros sistemas operativos que se han desarrollado en ocasiones en lenguaje ensamblador. La utilización de un lenguaje de alto nivel ha permitido la portabilidad del sistema a muchas máquinas diferentes. Lo mismo le ocurre a Linux, por ello era primordial que el código de las aplicaciones pudiera compilarse de manera transparente (o casi), sea cual sea la máquina y los dispositivos utilizados. De hecho, el transporte de Linux a otra máquina se resume en adaptar la parte del código que es específica de la máquina. Los módulos independientes de la arquitectura pueden reutilizarse tal cual.

- **Máquina virtual**, El sistema operativo ofrece una máquina virtual al usuario y al programas que ejecuta. Éste se ejecuta en una máquina física (hardware) que posee una interfaz de programación de bajo nivel, y proporciona abstracciones de alto nivel y una nueva interfaz de programación y uso más evolucionada. El sistema operativo el pues una interfaz entre las aplicaciones y la máquina. Por ello, es por lo que todas las tareas físicas (acceso a dispositivos externos o internos, a la memoria, etc.) se delegan en el sistema operativo. Esta encapsulación del hardware (y su diversidad) libera a los desarrolladores de la complejidad de gestionar todos los dispositivos existentes, ya que el sistema operativo es quien se encarga de ello. Esto también evita que el usuario se limite a una máquina, ya que si el sistema operativo está disponible sobre varias arquitecturas, la interfaz de usuario y de programación será la misma en todas.
- **Compartir el procesador (gestión de la CPU)**. Una de las características principales del sistema operativo UNIX es que el multitarea, es decir, vario programas (procesos = programas en ejecución) pueden ejecutarse al mismo tiempo. Para ello el sistema implementa una lista de procesos (que periódicamente ordena) para distribuir el procesador entre ellos de la mejor forma posible. Esta gestión debe ser muy sofisticada para no perjudicar a nadie, y se trata de uno de los mecanismos clave del sistema. En líneas generales, en Linux, a cada proceso se le atribuye un quantum de tiempo y elige un proceso a ejecutar durante ese quantum. Cuando ha transcurrido ese quantum, el sistema hace pasar al proceso al estado "listo para ejecutarse",y elige a otro proceso que se ejecuta durante otro quantum. El quantum es muy corto y el usuario tiene la impresión de que varios procesos se ejecutan simultáneamente, aunque sólo un proceso se ejecuta en un instante dado.
- **Gestión de la memoria**. El sistema se encarga de gestionar la memoria física de la computadora. En un entorno multiusuario y multitarea, el sistema debe efectuar una gestión muy rigurosa de la memoria. Como la memoria física de la máquina es a menudo insuficiente, el sistema utiliza entonces una parte del disco duro como memoria virtual (área de intercambio o *swap*). El sistema operativo debe ser capaz de gestionar hábilmente la memoria para poder satisfacer las peticiones de los distintos procesos tan deprisa como sea posible. Además, debe asegurar la protección de las zonas de memoria asignadas a los procesos para evitar modificaciones no autorizadas.

- **Gestión de recursos**. De manera general, el sistema operativo se encarga de gestionar los recursos disponibles (entre los cuales el procesador y la memoria son casos particulares e importantes). El sistema ofrece a los procesos una interfaz que permite compartir los recursos (discos, impresoras, etc.) del hardware. Implementa un sistema de protección que permite a los usuarios y a los administradores proteger el acceso a sus datos. El sistema mantiene listas de recursos disponibles y en curso de utilización, lo que permite atribuirlos a los procesos que los necesiten. El sistema operativo conoce en todo momento los procesos que utilizan los recursos de la máquina, y puede detectar así los conflictos de acceso.
- **Centro de comunicaciones de la máquina**. Una de las tareas del sistema operativo es la de gestionar los diferentes eventos procedentes del hardware (interrupciones) o de las aplicaciones (llamadas al sistema). Estos eventos son importantes, y el sistema debe tratarlos y, llegado el caso, enviarlos a los procesos afectados. Pero si el sistema operativo es capaz de responde a un evento, debe ser capaz también de comunicar varios procesos. De esta manera, los procesos podrán comunicarse entre sí, intercambiar información, sincronizarse, etc. UNIX pone a disposición del desarrollador varios mecanismo que van desde las señales a los IPC, pasando por las tuberías (pipes) y los sockets. Es por todo esto, por lo que el *kernel* del sistema operativo es el centro de comunicaciones de la máquina. Cuando dos procesos intercambian datos, es porque el sistema operativo implementa mecanismos sofisticados junto con recursos específicos.
- **Modo *kernel* y modo usuario**. Un proceso en un sistema operativo UNIX posee dos niveles de ejecución: *kernel* y usuario. El modo *kernel* constituye un modo privilegiado de ejecución; en este modo no se imponen ninguna restricción al *kernel* del sistema, y este último puede utilizar todas las instrucciones del procesador, manipular toda la memoria, dialogar directamente con todos los controladores de dispositivos, etc. El modo usuario es el modo de ejecución normal de un proceso; en este modo el proceso no posee ningún privilegio: ciertas instrucciones están prohibidas, sólo tiene acceso a las zonas de memoria que se la han signado, y no puede interactuar con el hardware. Un proceso en modo usuario efectúa operaciones en su entorno, sin interferir con los demás procesos, y puede ser interrumpido en cualquier momento, no obstaculizando su funcionamiento. Las llamadas al sistema, son un concepto muy importante en un sistema operativo tipo UNIX. Un proceso que se ejecute en modo usuario no puede acceder directamente a los recursos de la máquina (hardware), para ello debe efectuar llamadas al sistema. Una llamada al sistema es una petición transmitida por un proceso al *kernel*. Este último trata la petición en modo *kernel*, con todos los privilegios, y envía los resultados al proceso que prodigue su ejecución normal. Bajo UNIX las llamadas al sistema provocan un cambio: el proceso ejecuta una instrucción del procesador que le hace pasar al modo *kernel*. A continuación ejecuta una función de tratamiento vinculada a la llamada al sistema que ha efectuado, y luego vuelve al modo usuario para proseguir su ejecución. De este modo, el propio proceso trata su llamada al sistema, ejecutando una función del *kernel*. Esta función se supone que es fiable y puede ejecutarse en modo *kernel*, contrariamente a la función ejecutada por el proceso en modo usuario.

1.4.2. Descripción de Linux y de sus Funcionalidades.

- Linux se diseñó inicialmente como un clónico de UNIX distribuido libremente que funciona en máquinas PC con procesador 386, 486 o superior. Aunque se diseñó inicialmente para la arquitectura I386, en la actualidad funciona sobre otras plataformas como los procesadores Compac Alpha AXM, Sun Sparc, DEC VAX, ciertas plataformas basadas en los Motorola 68000 como Amiga y Atari, las máquinas de tipo MIPS, PowerPC64, ARM, Intel IA-64, AMD x86-64, Cris, etc.
- Linux es una implementación de UNIX que respeta la especificación POSIX pero que posee también ciertas extensiones propias de las versiones System V y BSD de UNIX. Esto implica la adaptación del código de aplicaciones desarrolladas inicialmente para otros sistemas UNIX. El término POSIX significa Portable Operating System Interface. Se trata de documentos producidos por IEEE y estandarizados por ANSI y el ISO. El objetivo de POSIX es permitir tener un código fuente transportable.
- El código de Linux es un código original, que no es propietario en absoluto y cuyos programas en código fuente se distribuyen libremente bajo cobertura de licencia GPL, que es la licencia pública general de GNU.

- Las funcionales de este sistema operativo son múltiples y corresponden a la idea que puede hacerse de un sistema UNIX moderno:
  - Sistema operativo de estilo UNIX. Compatible POSIX.
  - Multitarea, multiprocesador. Puede ejecutar programas al mismo tiempo, ya sea con uno o varios procesadores.
  - Multiplataforma. Puede funcionar en múltiples arquitecturas hardware.
  - Multiusuario. Como en todo sistema UNIX, Linux permite trabajar a varios usuarios simultáneamente en la misma máquina.
  - Protección de memoria entre procesos.
  - Soporte de comunicaciones interprocesos (pipes, IPC, sockets).
  - Gestión de diferentes señales.
  - Gestión de terminales según la norma POSIX. Linux, proporciona también los pseudoterminales y los controles de procesos.
  - Soporte de un gran número de dispositivos ampliamente extendidos (tarjetas de sonido, gráficas, de red, SCSI, etc.).
  - Buffer caché. Zona de memoria intermedia para las entradas/salidas de los diferentes procesos.
  - Gestión de memoria (memoria virtual). Una página sólo se carga si es necesaria en memoria.
  - Librerías compartidas y dinámicas. Las librerías dinámicas sólo se cargan cuando son necesarias y su código se comparte si varias aplicaciones las utilizan.
  - Soporte de múltiples sistemas de archivos. Sistemas de archivos que permiten gestionar tanto particiones Linux con el sistema de archivos Ext2, por ejemplo, como particiones en otros formatos (MS-DOS, iso9660, etc.)
  - Soporte de la familia de protocolos TCP/IP y de otros protocolos de red.
  - Capacidad de multiprocesamiento simétrico, de funcionamiento en "cluster",
  - Escrito en "C".
  - Posibilidad de depuración en tiempo de ejecución.
  - Carga de módulos en tiempo de ejecución.

1.4.3. Estructura General del Sistema Operativo Linux.

- Tal y como hemos visto, el sistema operativo se compone de varios elementos importantes. UNIX y Linux se han desarrollado de manera modular de modo que se pueden distinguir fácilmente las diferentes partes que lo componen. La ventaje de esta estructuración es que permite su modificación y mejora sin excesiva dificultad. La incorporación de ciertos elementos, como llamadas al sistema, controladores de dispositivos u otros, es sencilla y no obliga a rediseñar la estructura del sistema.
- De forma general los distintos elementos que nos encontramos en el *kernel* de Linux son los siguientes:
  - Llamadas al sistema. Implementación de operaciones que deben ejecutarse en modo *kernel.*
  - Sistema de archivo. Entradas/salidas de los dispositivos.
  - Buffer caché. Memoria intermedia sofisticada para entradas/salidas.
  - Controladores de dispositivos. Gestión a bajo nivel de discos, tarjetas, impresoras, etc.
  - Gestión de la red. Protocolos de comunicaciones en red.
  - Interfaz con la máquina. Código (generalmente en ensamblador) de acceso a bajo nivel al hardware.
  - Núcleo del *kernel.* Gestión de procesos (creación, duplicación, destrucción, etc.), gestor de órdenes, señales, módulos cargables (carga de ciertas partes del *kernel* cuando se requieren), gestión de memoria (gestión de la memoria física y la memoria virtual), etc.

La siguiente figura (Figura 1.9) representa la estructura del sistema operativo Linux, que es muy parecida a la del sistema operativo UNIX.

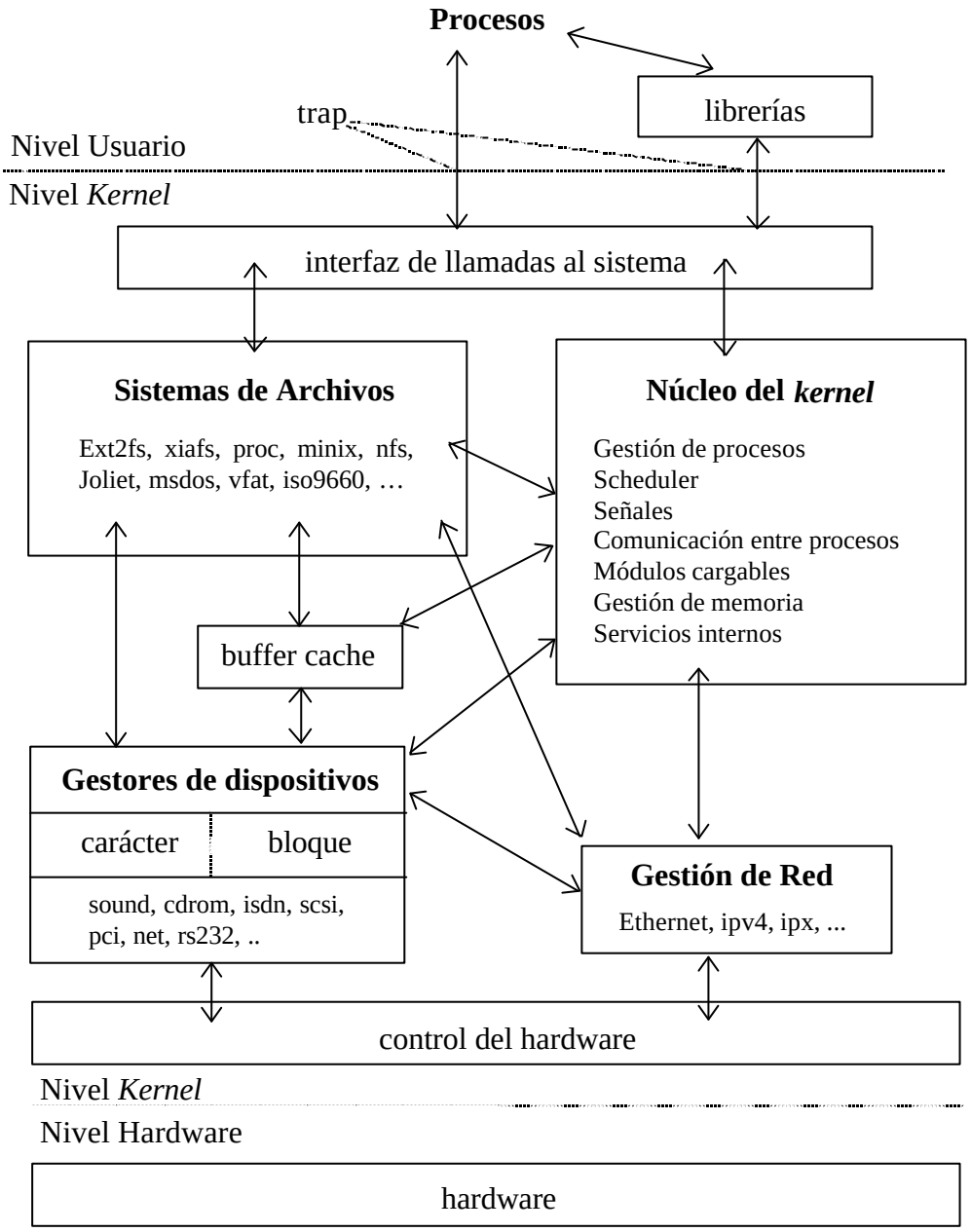

Figura 1.9. Estructura general del sistema operativo Linux *(kernel)*.

1.4.4. Organización del Código Fuente del *Kernel*.

- Los programas fuente del *kernel* de Linux se instalan normalmente en el directorio */usr/scr/linux*. En realidad, éste se puede obtener vía *ftp* desde *ftp://ftp.kernel.org/pub/linux/kernel/*, están empaquetados en un archivo *tar* comprimido (.tgz), que hay que extraer bajo el directorio */usr/scr/*, creándose el directorio */usr/scr/linux*.

- Se organizan de manera jerárquica, como se puede observar en la figura 1.10. Cada directorio o subdirectorio se dedica a ciertas funciones del *kernel*:
  - *Documentación*. Un cierto número de archivos de documentación respecto a la configuración del *kernel* o el funcionamiento de ciertos móculos.
  - *include*. Todos los archivos de cabecera necesarios para la generación del *kernel*, pero también para la compilación de aplicaciones.
  - *fs*. Sistema de archivos.
  - *init*. El *main.c* de Linux.
  - *ipc*. Gestión de la comunicación interproceso según la norma System V.
  - *mm*. Gestión de memoria.
  - *kernel*. Principales llamadas al sistema.
  - *lib*. Módulos diversos.
  - *drivers*. Controladores de dispositivos.
  - *net*. Protocolos de red.
  - *arch*. Código dependiente de la plataforma.
  - *scripts*. Scripts utilizados para la configuración y para la generación de las dependencias del *kernel*.

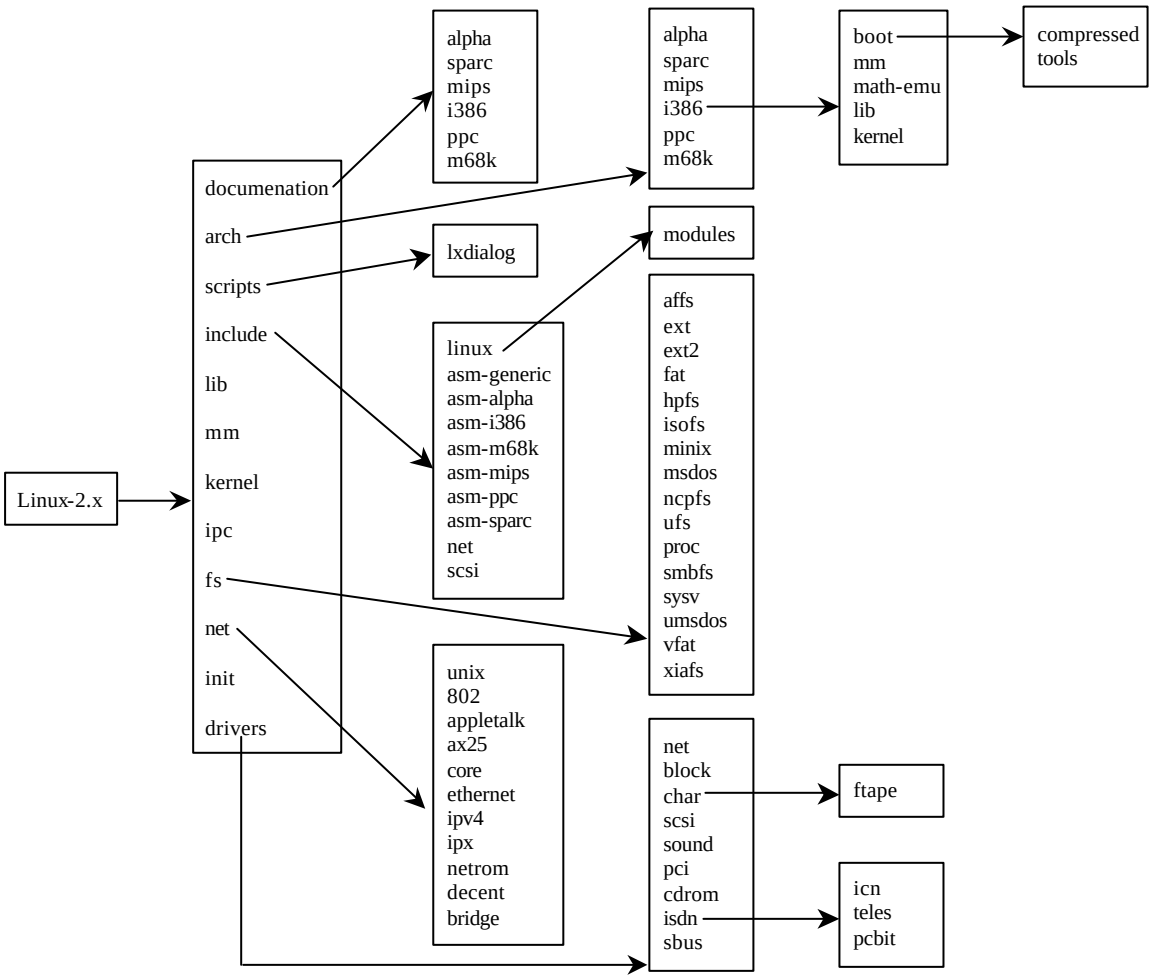

Figura 1.10. Organización de código fuente (árbol de directorios) del *kernel*.

- Los archivos de cabecera utilizados por el preprocesador del lenguaje C, se sitúan en el directorio /usr/include, y definen la interfaz de las librerías utilizadas por los programas que se compilan. Estos archivos de cabecera se enlazan con la librería C, que se distribuye independientemente del *kernel*.

- El *kernel* de Linux dispone también de archivos cabecera que se utilizan en su compilación. Estos archivos se sitúan en el directorio */usr/scr/linux/include*, en donde encontraremos principalmente dos subdirectorios: (1) *linux*: Este directorio contiene las declaraciones independientes de la arquitectura; (2) *asm*: este directorio contiene las declaraciones dependientes de la arquitectura (se trata de un enlace con otro directorio (por ejemplo, *asm-i836* para la arquitectura i386))
- Dos enlaces simbólicos permiten acceder también a estos archivos desde el directorio /usr/include. También es posible incluir en un programa archivos de cabecera que definen constantes y tipos utilizados por el *kernel* utilizando las expresiones <*asm/file.h*> y <*linux/file.h*>. Hay que observar, sin embargo, que no es aconsejable la inclusión de dichos archivos, es preferible incluir los archivos estándar de la librería C, porque están preparados para ello. Estos archivos cabecera se encargan de importar las definiciones del *kernel*.

1.4.5. Funcionamiento General del *Kernel* de Linux.

Sabemos que un proceso se ejecuta normalmente en modo usuario y debe pasar a modo *kernel* para ejecutar las llamadas al sistema, pero nos podemos preguntar ¿cómo se implementa una llamada al sistema?.

1.4.5.1. Implementación de una Llamada 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 <*asm/unistd.h*>. Por ejemplo, la llamada al sistema *open* tiene el número 5. El nombre se define en el archivo ensamblador *arch/i386/kernel/entry.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 *arch/i386/kernel/entry.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 volver de esta función, *system_call* vuelve a quien ha llamado, este retorno vuelve a pasar al proceso a modo usuario.

1.4.5.2. Creación de una Llamada 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 (por ejemplo, la versión 2.0 de Linux contaba con 164 llamadas al sistema, desde la 0 a la 163), y después hay que añadir el primer número libre en el archivo *include/asm-i386/unistd.h*:

```
/* Llamadas al sistema del kernel */
#define __NR_sched_get_priority_min 160
#define __NR_sched_rr_get_interval 161
#define __NR_nanosleep 162
#define __NR_mremap 163
/* Llamada al sistema añadida */
#define __NR_show_mult 164
```

- A continuación, en el archivo *arch/i386/kernel/entry.S* (o bien en el archivo entry.S de la arquitectura correspondiente) el puntero a la función *show_mult* que implementa la llamada al sistema.

```
/* Llamadas al sistema del kernel */
.long SYMBOL_NAME(sys_sched_get_priority_min) /* 160 */
.long SYMBOL_NAME(sys_sched_rr_get_interval) /* 161 */
.long SYMBOL_NAME(sys_nanosleep) /* 162 */
.long SYMBOL_NAME(sys_mremap) /* 163 */
/* Valor inicial del número de llamadas */
/* .space (NR_syscalls-162)*4 */
/* Llamada al sistema añadida */
.long SYMBOL_NAME(sys_show_mult) /* 164 */
.space (NR_syscalls-164)*4 /* Actualizar el número NR_syscalls */
```

- En este momento, se efectúan todos los enlaces que permiten la utilización de la llamada al sistema y entonces sólo queda implementar dicha función. El código de la llamada al sistema puede estar integrado en el archivo *kernel/sys.c* que agrupa un cierto número de llamadas al sistema. El código añadir es el siguiente:

```
asmlinkage int sys_show_mult(int x, int y, int *res)
{
        int error;
        int compute;
        /* Verificación de la validez de la variable res */
        error = verify_area(VERIFY_WRITE, res, sizeof(*res));
        if (error)
                return error;
        /* Cálculo del resultado de la multiplicación */
        compute = x*y;
        /* Copia el resultado en la memoria del usuario */
        put_user(compute, res);
        printf("Value computed: %d * %d = %d \n", x, y, compute);
        /* Llamada al sistema finalizada */
        return 0;
}
```

- Una vez recompilado 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 *<syscall.h>*. En el caso de una llamada al sistema que acepta tres parámetros, hay que utilizar la macroinstrucción *_syscall3*:

```
#include <stdio.h>
#include <stdlib.h>
#include <linux/unistd.h>
_syscall3 (int showmult(int x, int y, int *result);
main ()
{
        int ret = 0;
        show_mult(2, 5, &ret);
        printf((“Resultado: %d * %d = %d \n”, 2, 5, ret);
}
```

- La macroinstrucción *_syscall3* es expandida por el preprocesador y proporciona el código siguiente:

```
int show_mult(int x, int y, int *result)
{
        long _res;
        _asm_ _volatile (“int $0x80”
                : “=a (_res)
                : “0” (164)
                “b” ((long)(x)),
                “c” ((long)(y)),
                “d” ((long)(result)));
        if (_res>=0)
                return (int)_res;
        errno=_res;
        return –1;
}
```

- La función *show_mult* es generada por _syscall3, que inicializa los registros del procesador (el número de la llamada al sistema se coloca en el registros 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 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.

1.4.5.3. Códigos de Retorno.

- El *kernel* puede detectar errores en la ejecución de una llamada al sistema. En este caso, se devuelve un código de error al proceso que hace la llamada. Generalmente, se devuelve el valor –1 en caso de error. Con el objetivo de permitir al proceso que ha llamado determinar la causa del error, la librería C proporciona la variable global errno. Esta variable se actualiza tras cada llamada al sistema que cause un error, y contiene un código que indica la causa del error. No se actualiza tras una llamada al sistema con éxito, por lo que hay que comprobar el valor de retorno de cada llamada al sistema y utilizar el valor de errno sólo en caso de fallo. El archivo de cabecera *<errno.h>* define numerosas constantes que presentan los posibles errores. La librería C proporciona también las dos variables siguientes:

```
extern int _sys_nerr;
extern char *_sys_errlist[];
```

- La variable _sys_nerr contiene el número de códigos de error implementados. La tabla _sys_errlist contiene a su vez las cadenas de caracteres que describen todos los errores. El mensaje de error correspondiente al último error detectado puede obtenerse por medio de _sys_errlist[errno]. Además, la librería C proporciona dos funciones:

```
#include <stdio.h>
#include <errno.h>
void perror (const char *s);
char *strerror(int errnum);
```

- La función perror muestra un mensaje en la salida de errores del proceso que llama. Este mensaje contiene la cadena especificada por el parámetro s y el mensaje de error correspondiente al valor de errno. La función strerror devuelve la cadena de caracteres correspondiente al error cuyo código se especifica en el parámetro errnum.
- También es interesante destacar que el *kernel* contiene numerosas funciones de utilidad para la manipulación del espacio de direccionamiento. El espacio de direccionamiento en modo *kernel* es diferente de su espacio de direccionamiento en modo usuario. Por tanto, resulta imposible acceder directamente a los datos cuya dirección se ha pasado como parámetro a una llamada al sistema. Linux proporciona varias funciones para acceder a estos datos, entre ellas podemos destacar la siguientes (para mas información utilizar *man nombre_funcion*):

```
int verify_area(int type, const void *addr, unsigned long size);
void put_user(unsigned long valur, void *addr);
void memcpy_tofs(void *to, const void *from, unsigned long size);
void memcpy_fromfs(void *to, const *from, unsigned long size);
void *kmolloc(unsigned int size, int priority);
void kfree(void *addr);
void vmallos(unsigned long size);
void vfree(void *addr);
```