SO 22-23: Bibliografía Tema 1

Stallings

Tarea 1.1: Desde la sección 2.6 hasta el final del capítulo 2

2.6. Sistemas Unix tradicionales

Historia

En 1970 se desarrolla UNIX en los laboratorios Bell. Llamado una "versión recortada de Multics", aunque muy influenciado por CTSS. Se pudo portear entre dos sistemas distintos (PDP7 a PDP11), lo cual fue el primer indicio de que sería un sistema para cualquier máquina.

El mayor hito fue cuando se pudo reescribir en C: hasta ese momento se creía que un SO, que maneja eventos críticos de tiempo, sería imposible de escribir en otra cosa que no fuera ensamblador. El salto a un lenguaje de alto nivel fue un cambio muy positivo.

El sistema ganó popularidad en los laboratorios Bell y, cuando se publicó en una revista técnica en 1974 se empezaron a distribuir licencias a instituciones comerciales y universidad, lo que generó una gran cantidad de variantes y mejoras.

Descripción

La imagen da una descripción general de la arquitectura UNIX: el hardware subyacente es gestionado por el software del sistema operativo. El SO se denomina frecuentemente núcleo o kernel, para destacar su aislamiento del usuario y las aplicaciones. Es a lo que nos referiremos como UNIX. Pero UNIX incluye servicios de usuario e interfaces, que podemos agrupar en el shell (otro software de interfaz), y los componentes del compilador C (compilador, ensamblador, cargador). La capa externa está formada por las aplicaciones de usuario y la interfaz de usuario del compilador C.

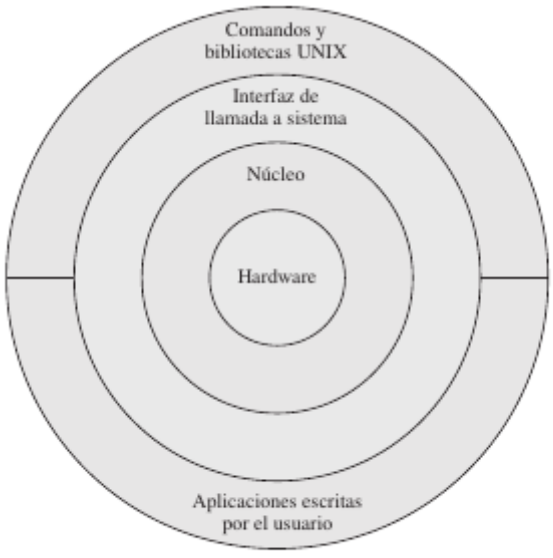

Figura 2.14. Arquitectura general de UNIX.

La siguiente imagen es una vista más cercana al kernel. Los programas de usuario pueden invocar servicios del SO directamente, o a través de bibliotecas. La interfaz de llamadas al sistema es la frontera del usuario, y permite que las aplicaciones de alto nivel tengan acceso a funciones del núcleo.

Por el otro lado, el SO tiene rutinas primitivas que interactúan directamente con el hardware.

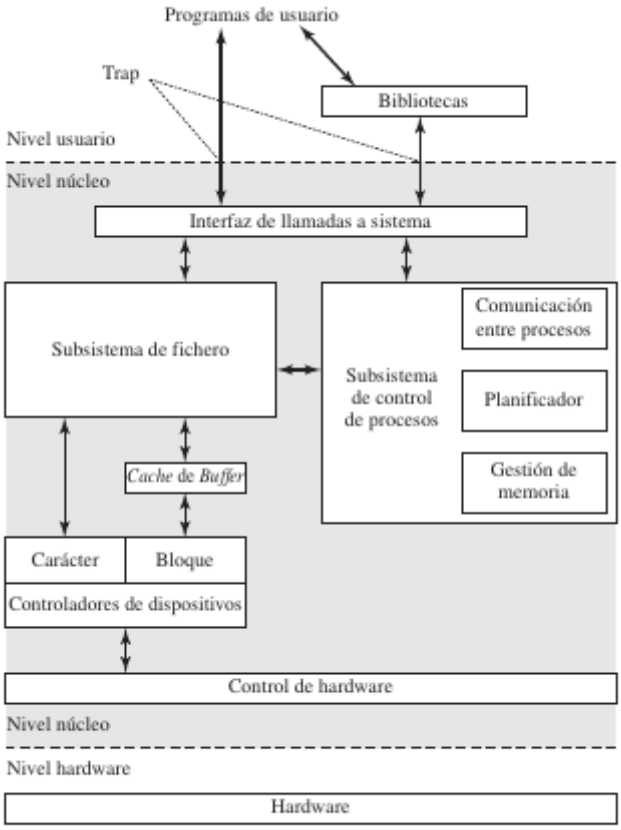

Figura 2.15. Núcleo tradicional de UNIX [BACH86].

Entonces, con las dos interfaces que hemos comentado, vemos que el sistema se divide en dos grandes bloques: control de procesos, y gestión de ficheros y E/S.

- El subsistema de control de procesos se encarga de:
  - Gestión de memoria
  - Planificación y ejecución de procesos
  - Sincronización y comunicación de procesos
- El subsistema de ficheros:
  - Intercambia datos entre la memoria y los dispositivos externos, de flujos de caracteres y de bloques. Para ello usa variedad de controladores de dispositivos
    - Para transferencias de bloques, se usa una técnica de caché de discos: entre el espacio de direccionamiento del usuario y el del dispositivo externo, se interpone un buffer de sistema en memoria principal.

Todo lo anterior lo hemos dicho refiriéndonos a sistemas UNIX tradicionales. Sobre los sistemas tradicionales:

1. Se diseñaron para ejecutar sobre un único procesador, y carecen de capacidad para proteger sus estructuras de datos en caso de acceso concurrente de múltiples procesadores.

2. Su núcleo no es muy versátil, soporta un único tipo de sistema de archivos, una única política de planificación de procesos y un único formato de fichero ejecutable
3. Su núcleo no está diseñado para ser extensible y tiene pocas utilidades para la reutilización de código. En consecuencia, cuando se quiso expandir hubo que añadir mucho código; quedando un núcleo de gran tamaño y no modular.

2.7. Sistemas Unix modernos

Cuando UNIX evolucionó, un gran número de versiones surgió, cada una de ellas con ciertas características útiles. Fue necesario implementar una nueva versión que aunara las innovaciones más importantes, añadiera otras características de diseño de los SO modernos, y produjera una arquitectura más modular.

La arquitectura de la imagen 2.16 muestra los aspectos típicos de un núcleo UNIX moderno. Existe un pequeño núcleo de utilidades escritas modularmente, que proporciona funciones y servicios imprescindibles para procesos del SO. Cada círculo externo representa funciones y una interfaz que podría implementarse de varias formas.

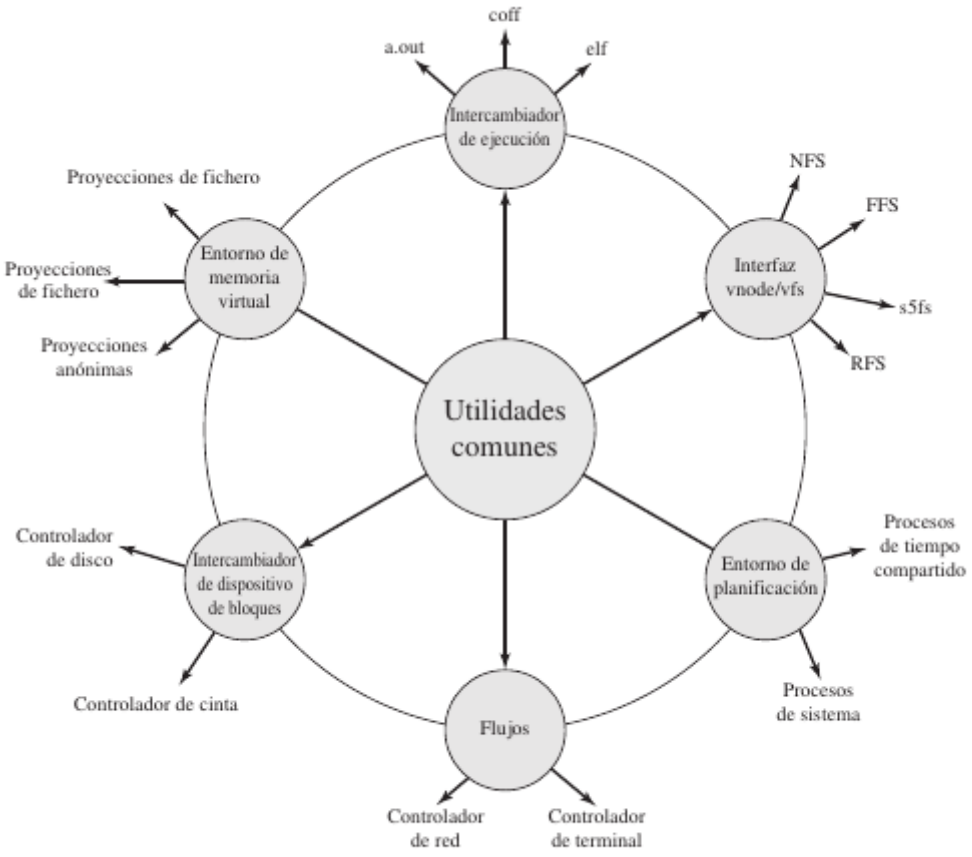

Figura 2.16. Núcleo UNIX moderno [VAHA96].

System V Release 4 (SVR4)

SVR4, desarrollado conjuntamente por AT&T y Sun Microsistemas, fue casi una reescritura completa del núcleo del System V y produjo una implementación bien organizada pero compleja. Las nuevas características incluyen:

- Soporte al procesamiento en tiempo real
- Clases de planificación de procesos
- Estructuras de datos dinámicamente asignadas
- Gestión de memoria virtual
- Sistema de ficheros virtual

- Núcleo expulsivo

SVR4 mezcló la visión comercial y académica, y se desarrolló para dar una plataforma uniforme que permitiera el despliegue comercial del UNIX. Lo logró es una de las variantes más importantes del sistema: incorpora la mayoría de características importantes desarrolladas de forma integrada y comercialmente viable. Ejecuta en un gran rango de máquinas, desde 32 bits a supercomputadores, y se usa como ejemplo a lo largo de este libro.

Solaris 9

Solaris es una versión UNIX de Sun basada en SVR4. La última versión es la 9, y proporciona todas las características de SVR4 más un conjunto de características avanzadas, como:

- Núcleo multihilo completamente expulsivo, con soporte completo para SMP
- Interfaz orientada a objetos para los sistemas de ficheros

Solaris es la implementación UNIX más utilizada y comercialmente más exitosa. Para algunas características de los sistemas operativos, se utiliza a Solaris como ejemplo en este libro.

4.4BS

Las series de UNIX BSD (Berkeley Software Distribution) han jugado un papel importante en el desarrollo de la teoría de diseño de los sistemas operativos. 4.xBSD se ha utilizado ampliamente en instalaciones académicas y ha servido como base de algunos productos comerciales UNIX. Seguramente BDS es responsable de gran parte de la popularidad de UNIX, y la mayoría de las mejoras de UNIX aparecieron en primer lugar en las versiones BSD.

4.4BSD fue la versión final de BSD que Berkeley produjo, disolviéndose luego la organización encargada del diseño e implementación. Se trata de una actualización importante de 4.3BSD, que incluye, entre muchas otras:

- Nuevo sistema de memoria virtual
- Cambios en la estructura del núcleo

La última versión del sistema operativo de Macintosh, Mac OS X, se basa en 4.4BSD.

2.8. LINUX

Historia

Linux comienza como una variante UNIX para la arquitectura de un PC IBM. Linus Torvald, estudiante finlandés de informática, escribió la versión inicial y la distribuyó en Internet en 1991. Desde entonces, usuarios de internet han colaborado en su desarrollo: al ser libre y de código abierto, se convirtió en la alternativa a Sun e IBM. Hoy día, Linux es un sistema UNIX completo que ejecuta en casi cualquier plataforma.

La clave de su éxito está en la disponibilidad de los paquetes de software libre bajo la FSF (Free Software Foundation), que se centra en el software estable, multiplataforma, de calidad, por y para la comunidad de usuarios. El proyecto GNU proporciona herramientas a los desarrolladores software, así como la licencia pública GPL. Linus usó herramientas GNU para desarrollar el kernel, y lo distribuyó bajo licencia GPL, así que las distribuciones Linux actuales son producto del proyecto GNU de la FSF, los esfuerzos de Linus y la colaboración de muchos usuarios.

Además de su uso por muchos programadores individuales, Linux ha hecho una penetración significativa en el mundo corporativo. Esto no es sólo debido al software libre, sino también a la calidad del núcleo de Linux. Muchos programadores con talento han contribuido a la versión actual, dando lugar a un producto técnicamente impresionante. Más aún, Linux es muy modular y fácilmente configurable. Resulta óptimo para incrementar el rendimiento de una variedad de plataformas hardware. Además, con el código fuente disponible, los distribuidores pueden adaptar las aplicaciones y facilidades para cumplir unos requisitos específicos.

A lo largo de este libro, se proporcionarán detalles internos del núcleo de Linux.

Estructura modular

La mayoría de los núcleos Linux son monolíticos. Un núcleo monolítico incluye prácticamente toda la funcionalidad del SO en un gran bloque de código, que ejecuta como un único proceso con un único espacio de direccionamiento. Todos los componentes funcionales del núcleo tienen acceso a todas las estructuras internas de datos y rutinas. Si se hacen cambios sobre el sistema monolítico, todos los módulos y rutinas se tienen que reinstalar y reenlazar, y reiniciar el sistema para hacer efectivos los cambios. Como resultado, cualquier modificación es difícil - esto es particularmente cierto para Linux porque su desarrollo es global, por devs independientes asociados de forma difusa.

Linux no usa la técnica microkernel, pero consigue muchas ventajas de esta mediante su particular arquitectura modular. Linux está estructurado como una colección de **módulos cargables**, que pueden (algunos) cargarse o descargarse automáticamente bajo demanda. Un módulo es un fichero cuyo código puede enlazarse o desenlazarse del núcleo en tiempo real, que implementa funciones específicas: sistema de ficheros, controlador de dispositivo... Aunque pueda crearlos, un módulo no se ejecuta como su propio proceso o hilo, sino que se ejecuta en modo kernel en nombre del proceso actual.

Por tanto, aunque Linux se puede considerar monolítico, su estructura modular elimina algunas de las dificultades para desarrollar y evolucionar el núcleo. Los módulos cargables de Linux tienen dos características importantes:

- **Enlace dinámico:** un módulo de núcleo puede cargarse y enlazarse al núcleo mientras el núcleo está en memoria y ejecutándose. Un módulo también puede desenlazarse y eliminarse de la memoria en cualquier momento.
- **Módulos apilables:** los módulos se gestionan como una jerarquía. Los módulos individuales sirven como bibliotecas cuando los módulos cliente los referencian desde la parte superior de la jerarquía, y actúan como clientes cuando referencian a módulos de la parte inferior de la jerarquía.

El enlace dinámico facilita la configuración y reduce el uso de la memoria del kernel. En Linux, un programa de usuario o usuario puede cargar o descargar explícitamente módulos usando los comandos `insmod, rmmod`. El propio kernel detecta la necesidad de funciones particulares y carga/descarga los módulos pertinentes. Con módulos apilables, podemos definir dependencias entre módulos. Esto tiene dos ventajas:

1. El código común para un cjto de módulos similares (ej. controladores de un hardware similar) se puede mover a un único módulo, reduciendo la aplicación
2. El núcleo puede asegurar que los módulos necesarios están presentes, impidiendo descargar un módulo del cual otros módulos ejecutando dependen, y cargando módulos adicionalmente requeridos cuando se carga un nuevo módulo.

La figura 2.17 muestra un ejemplo de las estructuras utilizadas por Linux para gestión de módulos. La figura muestra la lista de los módulos del núcleo que existen después de que sólo dos módulos hayan sido cargados: FAT y VFAT. Cada módulo se define por dos tablas:

- **La tabla de módulos:** incluye los siguientes elementos:
  - *next: puntero al sig módulo. Todos los módulos se organizan en una lista enlazada. La lista comienza en un pseudo-módulo, no listado en la imagen.
  - *name: puntero al nombre del módulo.
  - size: tamaño del módulo en páginas de memoria.
  - usecount: contador de uso del módulo. El contador se incrementa cuando una operación realizada con las funciones del módulo comienza, y se decrementa cuando ésta finaliza.
  - flags: opciones del módulo.
  - nsyms: número de símbolos exportados.
  - ndeps: número de módulos referenciados.
  - *syms: puntero a la tabla de símbolos del módulo.
  - *deps: puntero a la lista de módulos referenciados por este módulo.
  - *refs: puntero a la lista de módulos que usa el módulo.
- **La tabla de símbolos:** la tabla de símbolos define aquellos símbolos controlados por este módulo que se utilizan en otros sitios.

La Figura 2.17 muestra que el módulo VFAT se carga después del módulo FAT y que el módulo VFAT es dependiente del módulo FAT.

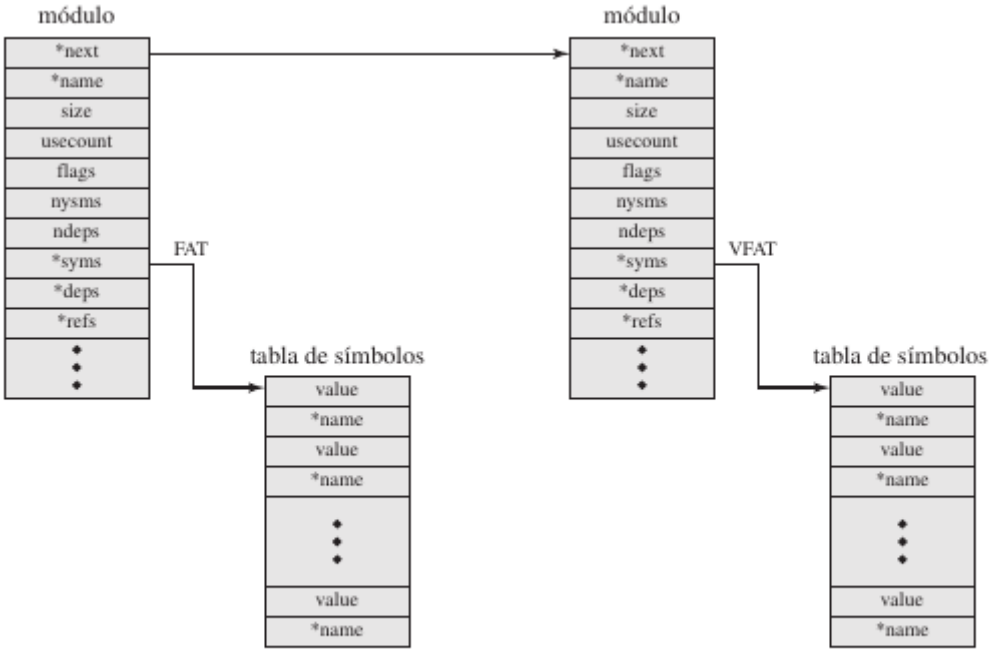

Figura 2.17. Lista ejemplo de módulos de núcleo de Linux.

Componentes del núcleo

La Figura 2.18, muestra los principales componentes del núcleo Linux tal y como están implementados en una arquitectura IA-64 (por ejemplo, Intel Itanium). La figura muestra varios procesos ejecutando encima del núcleo. Cada caja indica un proceso separado, mientras que cada línea curvada con una cabeza de flecha representa un *hilo de ejecución.

> *En Linux, no hay distinción entre los conceptos de proceso e hilo. Sin embargo, múltiples hilos en Linux se pueden agrupar de tal forma que, efectivamente, pueda existir un único proceso compuesto por múltiples hilos. Estos aspectos se discuten en el Capítulo 4.

El núcleo mismo está compuesto por una colección de componentes que interaccionan, usando flechas para indicar las principales interacciones. También se muestra el hardware subyacente como un conjunto de componentes utilizando flechas para indicar qué componentes del núcleo utilizan o controlan qué componentes del hardware. Todos los componentes del núcleo, por supuesto, ejecutan en la CPU, pero por simplicidad no se muestran estas relaciones.

Brevemente, los principales componentes del núcleo son los siguientes:

- **Señales**. El núcleo utiliza las señales para llamar a un proceso. Por ejemplo, las señales se utilizan para notificar ciertos fallos a un proceso como por ejemplo, la división por cero.
    - Ejemplos de señales:

        > - SIGHUP Desconexión de un terminal
        > - SIGCONT Continuar
        > - SIGQUIT Finalización por teclado
        > - SIGTSTP Parada por teclado
        > - SIGTRAP Traza
        > - SIGTTOU Escritura de terminal
        > - SIGBUS Error de bus
        > - SIGXCPU Límite de CPU excedido
        > - SIGKILL Señal para matar
        > - SIGVTALRM Reloj de alarma virtual
        > - SIGSEGV Violación de segmentación
        > - SIGWINCH Cambio de tamaño de una ventana
        > - SIGPIPE Tubería rota
        > - SIGPWR Fallo de potencia
        > - SIGTERM Terminación
        > - SIGRTMIN Primera señal de tiempo real
        > - SIGCHLD Cambio en el estado del hijo
        > - SIGRTMAX Última señal de tiempo real

- **Llamadas al sistema**. La llamada al sistema es la forma en la cual un proceso requiere un servicio de núcleo específico. Hay varios cientos de llamadas al sistema, que pueden agruparse básicamente en seis categorías:
    - Sistema de ficheros

        > - close Cierra un descriptor de fichero.
        > - link Construye un nuevo nombre para un fichero.
        > - open Abre y posiblemente crea un fichero o dispositivo.
        > - read Lee un descriptor de fichero.
        > - write Escribe a través de un descriptor de fichero.

    - Procesos

        > - execve Ejecuta un programa.
        > - exit Termina el proceso que lo invoca.
        > - getpid Obtiene la identificación del proceso.

- setuid Establece la identidad del usuario del proceso actual.
- prtrace Proporciona una forma por la cual un proceso padre puede observar y controlar la ejecución de otro proceso, examinar y cambiar su imagen de memoria y los registros.

- Planificación

  - sched_getparam Establece los parámetros de planificación asociados con la política de planificación para el proceso identificado por su pid.
  - sched_get_priority_max Devuelve el valor máximo de prioridad que se puede utilizar con el algoritmo de planificación identificado por la política.
  - sched_setscheduler Establece tanto la política de planificación (por ejemplo, FIFO) como los parámetros asociados al pid del proceso.
  - sched_rr_get_interval Escribe en la estructura timespec apuntada por el parámetro tp el quantum de tiempo round robin para el proceso pid.
  - sched_yield Un proceso puede abandonar el procesador voluntariamente sin necesidad de bloquearse a través de una llamada al sistema. El proceso entonces se moverá al final de la cola por su prioridad estática y un nuevo proceso se pondrá en ejecución.

- Comunicación entre procesos

  - msgrcv Se asigna una estructura de buffer de mensajes para recibir un mensaje. Entonces, la llamada al sistema lee un mensaje de la cola de mensajes especificada por msqid en el buffer de mensajes nuevamente creado.
  - semctl Lleva a cabo la operación de control especificada por cmd en el conjunto de semáforos semid.
  - semop Lleva a cabo operaciones en determinados miembros del conjunto de semáforos semid.
  - shmat Adjunta el segmento de memoria compartido identificado por shmid al segmento de datos del proceso que lo invoca.
  - shmctl Permite al usuario recibir información sobre un segmento de memoria compartido, establecer el propietario, grupo y permisos de un segmento de memoria compartido o destruir un segmento.

- Socket (red)

  - bind Asigna la dirección IP local y puerto para un socket. Devuelve 0 en caso de éxito y -1 en caso de error.
  - connect Establece una conexión entre el socket dado y el socket asociado remoto con sockaddr.
  - gethostname Devuelve el nombre de máquina local.
  - send Envía los bytes que tiene el buffer apuntado por *msg sobre el socket dado.
  - setsockopt Envía las opciones sobre un socket.

- Misceláneos

  - create_module Intenta crear una entrada del módulo cargable y reservar la memoria de núcleo que será necesario para contener el módulo.
  - fsync Copia todas las partes en memoria de un fichero a un disco y espera hasta que el dispositivo informa que todas las partes están en

almacenamiento estable.
- query_module Solicita información relacionada con los módulos cargables desde el núcleo.
- time Devuelve el tiempo en segundos desde 1 de enero de 1970.
- vhangup Simula la suspensión del terminal actual. Esta llamada sirve para que otros usuarios puedan tener un terminal «limpio» en tiempo de inicio.

- **Procesos y planificador**. Crea, gestiona y planifica procesos.
- **Memoria virtual.** Asigna y gestiona la memoria virtual para los procesos.
- **Sistemas de ficheros**. Proporciona un espacio de nombres global y jerárquico para los ficheros, directorios y otros objetos relacionados con los ficheros. Además, proporciona las funciones del sistema de ficheros.
- **Protocolos de red.** Da soporte a la interfaz Socket para los usuarios, utilizando la pila de protocolos TCP/IP.
- **Controladores de dispositivo tipo carácter.** Gestiona los dispositivos que requiere el núcleo para enviar o recibir datos un byte cada vez, como los terminales, los módems y las impresoras.
- **Controladores de dispositivo tipo bloque.** Gestiona dispositivos que leen y escriben datos en bloques, tal como varias formas de memoria secundaria (discos magnéticos, CDROM, etc.).
- **Controladores de dispositivo de red.** Gestiona las tarjetas de interfaz de red y los puertos de comunicación que permiten las conexiones a la red, tal como los puentes y los encaminadores.
- **Traps y fallos**. Gestiona los traps y fallos generados por la CPU, como los fallos de memoria.
- **Memoria física.** Gestiona el conjunto de marcos de páginas de memoria real y asigna las páginas de memoria virtual.
- **Interrupciones.** Gestiona las interrupciones de los dispositivos periféricos.

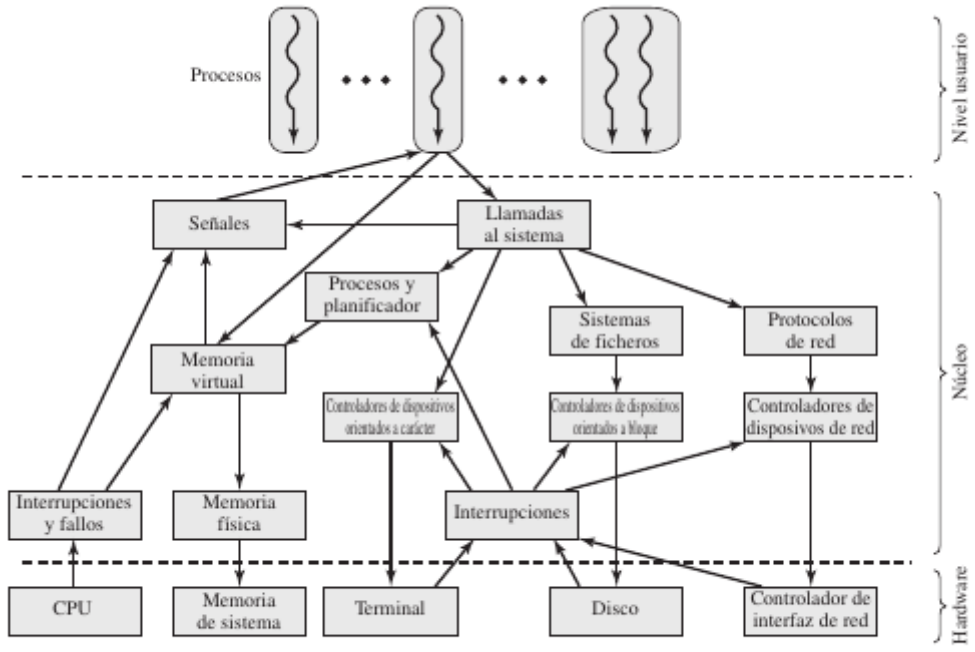

Figura 2.18. Componentes del núcleo de Linux.

Carretero

Tarea 1.1: Sección 2.9

2.9. Seguridad y protección

La seguridad consta de dos aspectos:

1. Garantizar la identidad de los usuarios -> AUTENTICACIÓN
2. Definir lo que cada usuario puede hacer -> PRIVILEGIOS

La seguridad es una de las funciones del SO que, para llevarla a cabo, se ha de basar en los mecanismos de protección proporcionados por el hardware.

Autenticación

Determinar que un usuario (sea persona, servicio o computadora) es quién dice ser. El SO dispone de un módulo de autenticación que se encarga de decidir la identidad de los usuarios. Actualmente el mecanismo más extendido son las contraseñas, pero no es el único.

Privilegios

Los privilegios especifican los recursos a los que tiene acceso cada usuario. Para simplificar la información de privilegios, es común organizar a los usuarios en grupos, y asignar privilegios a los grupos.

La información de los privilegios se puede asociar a recursos o a usuarios:

- **Información por recursos**: se asocia la ACL (access control list/lista de control de acceso) a cada recurso. Esta especifica los grupos y usuarios que pueden acceder a cierto recurso.
- **Información por usuario:** se asocia a cada usuario o grupo la lista de recursos que puede acceder, llamada lista de capacidades/capabilities.

Como hay muchas formas de acceder a un recurso, las listas de control de acceso y de capacidades tienen que especificar el **modo** de cada recurso (read, write, execute, delete, test, control, admin).

Los servicios relacionados con la seguridad y la protección se centran en la capacidad para asignar atributos de seguridad a los usuarios y a los recursos.

Silberchatz

Tarea 1.1: Secciones 2.7 y 2.8

2.7. Estructura de un SO

La ingeniería de un SO debe hacerse cuidadosamente para permitir que el sistema funcione correctamente y pueda ser modificado con facilidad. Un método habitual consiste en dividir la tarea en bloques más pequeños, en vez de tener un sistema monolítico.

Cada uno de los bloques (módulos) debe ser una parte bien definida del sistema, con entradas, salidas y funciones cuidadosamente establecidas. Veremos cómo los componentes más comunes de un SO se interconectan y funden en el kernel.

2.7.1. Estructura simple

Muchos sistemas no tienen una estructura bien definida: muchas veces, cuando se desarrollaron, iban a ser sistemas pequeños, simples y limitados, pero acabaron creciendo más allá de su ámbito original. Es el caso de MS-DOS, que al principio fue implementado por unas pocas personas que no esperaban que acabara siendo tan popular. Así que fue escrito para dar la máxima funcionalidad en el menor espacio posible, sin preocuparse mucho por una división en módulos.

En MS-DOS, las interfaces y los niveles de funcionalidad no están separados. Por ejemplo, **los programas de aplicación pueden acceder a las rutinas básicas de E/S** para escribir directamente en pantalla y en unidades de disco. Tal libertad hace a MS-DOS vulnerable ante programas erróneos o maliciosos, lo que hace que todo el sistema falle cuando los programas de usuario lo hacen. Como el Intel 8088 para el que fue escrito no proporciona modo dual ni protección hardware, **los diseñadores de MS-DOS no tuvieron más remedio que dejar accesible el hardware base.**

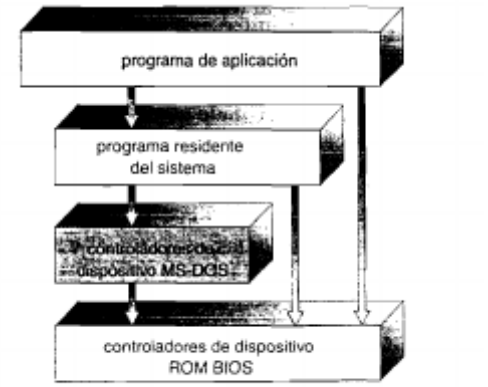

Figura 2.10 Estructura de niveles de MS-DOS.

Otro ejemplo de estructuración limitada fue el UNIX original, también limitado por su hardware. Cuenta con dos partes: el kernel y los programas del sistema. El kernel se divide en una serie de interfaces y controladores de dispositivo, que se han ido añadiendo y ampliando con los años hacia una estructura de niveles. Todo lo que está por debajo de la interfaz de llamadas al sistema y por encima del hardware es el kernel. El kernel proporcional el sistema de archivos, los mecanismos de planificación de la COU, la gestión de memoria y demás funciones del SO a través de llamadas al sistema. En resumen, es una enorme cantidad de funcionalidad que se combina en un sólo nivel, **una estructura monolítica difícil de implementar y mantener.**

Figura 2.11 Estructura del sistema UNIX.

2.7.2. Estructura en niveles

Con el soporte hardware adecuado, los SO pueden dividirse en partes más pequeñas y apropiadas que las mostradas en MS-DOS o UNIX. Esto da mayor control al SO sobre el ordenador y sobre las aplicaciones. Los desarrolladores tienen mayor libertad para cambiar el funcionamiento interno del sistema y crear un SO modular.

Un top-down approach permite **determinar las características y funcionalidades globales y separarlas en componentes. La ocultación de información a los niveles superiores da libertad al desarollador para implementar y cambiar las rutinas de bajo nivel** cuando y como quiera, siempre y cuando que la interfaz externa de la rutina sea la misma y funcione.

Un sistema puede modularizarse de muchas formas. Una es la **estructura de niveles o por capas**, en la que el SO se divide en capas: el nivel 0 es el hardware, y el nivel superior (N) es la interfaz de usuario. Un nivel de un SO es la implementación de un objeto abstracto, compuesto por datos (estructuras de datos) y por las operaciones que permiten manipular dichos datos (rutinas que los niveles superiores pueden invocar). A su vez, el nivel puede invocar las rutinas de niveles inferiores.

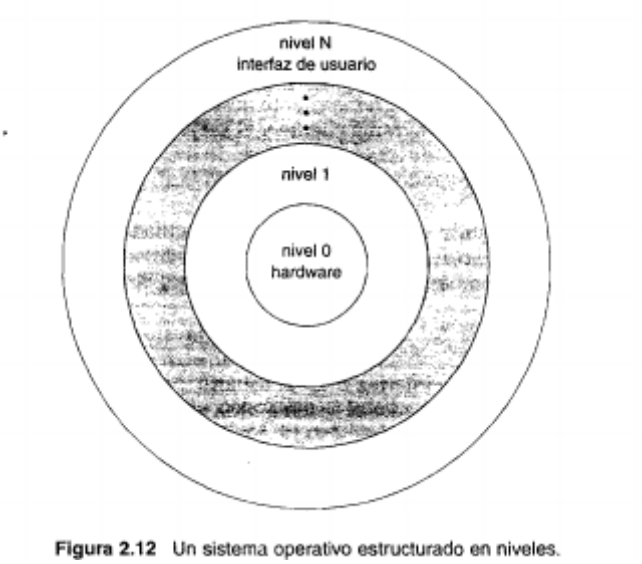

Figura 2.12 Un sistema operativo estructurado en niveles.

La principal ventaja de la estructura por niveles es la simplicidad de construcción y depuración. Cada nivel usa funciones y servicios del nivel inmediatamente inferior a él. El primer nivel sólo usa funciones del hardware que se suponen correctas. Así que, una vez depurado, se podrá suponer siempre su correcto funcionamiento. Cuando un nivel falla, sabemos que el problema se encuentra dentro del mismo o, en el peor de los casos, en los inferiores, pero suponemos que cada nivel ha sido depurado y asumido su buen funcionamiento antes de pasar al siguiente.

Cada nivel se implementa utilizando sólo las operaciones proporcionadas por los niveles infe riores. **Un nivel no necesita saber cómo se implementan dichas operaciones: sólo necesita saber qué hacen.** Por tanto, cada nivel oculta a los niveles superiores la existencia de determinadas estructuras de datos, operaciones y hardware.

Otro ejemplo menos obvio: normalmente, el controlador de almacenamiento de reserva está por encima del planificador CPU, dado que el controlador puede tener que esperar operaciones E/S, tiempo en el cual la CPU se asigna a otra tarea. **Pero, en sistemas de gran envergadura, la información sobre todos los procesos activos puede ser mayor que lo que cabe en memoria.** Así que el planificador de la CPU tiene que estar por encima del controlador de almacenamiento, porque necesita cargar y descargar desde el almacenamiento la información de procesos activos.

La principal dificultad es determinar los niveles, porque solo podremos usar los servicios de niveles inferiores. Esto requiere una planificación cuidadosa. Por ejemplo, el controlador del dispositivo de almacenamiento necesita estar en un nivel inferior a las rutinas de gestión de memoria, que requieren la capacidad de usar el almacenamiento.

Otro problema es que las implementaciones por niveles no **tienden a ser muy eficientes**: por cada nivel que bajamos, podemos modificar parámetros, pasar datos... lo cual crea una carga de trabajo adicional en la llamada al sistema que no existiría en otro tipo de estuctura. Esto ha creado cierta animosidad contra la estructura por niveles, y **los diseños más recientes tienden a incorporar un número cada vez menor de capas,** con más funcionalidad por nivel. Esto permite que aprovechemos la modularidad y a la vez evitemos los problemas de eficiencia y diseño de capas.

2.7.3. Microkernels

A medida que UNIX se expandía, el kernel crecía y se hacía más difíicl de gestionar. A mediados de los 80, en la Universidad de Carnegie Mellon se desarrolla Mach, un SO que modularizaba el kernel usando lo que llamaba microkernel. S**e eliminan todos los componentes no esenciales del kernel y se implementan como programas del sistema y de usuario, resultando en un kernel más pequeño.** No existe un consenso en qué programas deberían quedar fuera o dentro del kernel, pero normalmente los microkernels ofrecen una gestión de memoria y de procesos mínima, así como un mecanismo de comunicaciones.

La función principal del microkernel es proporcionar un mecanismo de comunicación (paso de mensajes) entre el programa cliente y los servicios que se ejecutan en el espacio de usuario. Por ejemplo, si el programa cliente quiere acceder a un archivo, debe interactuar con el servidor de archivos: estos dos nunca interactúan directamente, sino que lo hacen a través del microkernel.

Otra ventaja de los microkernels es la **facilidad para ampliar el SO**: todos los servicios nuevos se implementan en el espacio de usuario y no requieren modificar el kernel. Y cuando se modifica el kernel, es muy fácil por su reducido tamaño. **Tienen mayor portabilidad hardware, y son más seguros y fiables** porque, al ejecutarse la mayoría de cosas como procesos de usuario (no de kernel), un error no afecta al resto.

Varios sistemas operativos actuales son microkernel. Tru64 UNIX (antes Digital UNIX) proporciona una interfaz UNIX al usuario, pero se implementa con un kernel Mach. Éste transforma las llamadas al sistema UNIX en mensajes dirigidos a los servicios apropiados de nivel de usuario. Otro ejemplo es QNX: el microkernel de QNX proporciona servicios para paso de mensajes y planificación de procesos. También gestiona las comunicaciones de red de bajo nivel y las interrupciones hardware.

Por otro lado, **los microkernels pueden tener un peor rendimiento que otras estructuras debido a la carga de procesamiento generada por las funciones del sistema**. Windows NT fue un sistema microkernel con niveles, pero su rendimiento comparado a W95 fue bajo. Se solucionó parcialmente pasando varios niveles del espacio de usuario al espacio kernel, pero para cuando se diseñó WXP, el sistema era más de tipo monolítico que microkernel.

2.7.4. Módulos

Posiblemente la mejor técnica para diseñar SO es la que usa las ténicas de POO para crear un kernel modular. **En este caso, el kernel dispone de un conjunto de componentes fundamentales y enlaza dinámicamente los servicios adicionales**, bien durante el arranque o en tiempo de ejecución.

Tal estrategia usa módulos cargados dinámicamente y es habitual en implementaciones UNIX modernas, como Solaris, Linux y MacOS X. Por ejemplo, la estructura del SO Solaris se organiza alrededor de un kernel central con 7 tipos de módulo kernel cargables:

1. Clases de planificación
2. Sistemas de archivos
3. Llamadas al sistema cargables
4. Formatos ejecutables
5. Módulos STREAMS
6. Módulos misceláneos
7. Controladores de bus y de dispositivos

Un diseño así permite al kernel proporcionar servicios básicos y también permite implementar ciertas características dinámicamente. Por ejemplo, se pueden añadir al kernel controladores de bus y hardware específico, y puede agregarse como módulo cargable el soporte para diferentes sistemas de archivos. **El resultado es similar a un sistema por capas, en el sentido de que cada sección del kernel tiene interfaces bien definidas y protegidas, pero es más flexible que un sistema por niveles, porque cualquier módulo puede llamar a otro. Además es similar a la utilización de un microkernel porque el módulo principal sólo dispone de las funciones esenciales y de los conocimientos para cargar y comunicarse con otros módulos, pero es más eficiente porque los módulos no tienen que invocar el mecanismo de paso de mensajes.**

Mac OS x estructura el SO usando una estructura híbrida, mediante una técnica por niveles en la que uno de los niveles es el microkernel Mach.

Los niveles superiores incluyen los entornos de aplicación y un conjunto de servicios que proporcionan una interfaz gráfica a las aplicaciones. **Por debajo de estos niveles se encuentra el entorno kernel, que consta fundamentalmente del microkernel Mach y el kernel BSD.**

- **Mach** proporciona la gestión de memoria, el soporte para llamadas a procedimientos remotos (RPC, remote procedure call) y facilidades para comunicación interprocesos (IPC, interprocess comunication), así como un mecanismo de paso de mensajes y mecanismos de planificación de hebras de ejecución.
- **El módulo BSD** proporciona una interfaz de línea de comandos BSD, soporte para red y sistemas de archivos y una implementación de las API de POSIX, incluyendo Pthreads

Además de Mach y BSD, el entorno kernel proporciona un kit E/S para el desarrollo de controladores de dispositivo y módulos dinámicamente cargables (que Mac OS X denomina extensiones del kernel). Como se muestra en la figura, las aplicaciones y los servicios comunes pueden usar directamente las facilidades Mach y BSD.

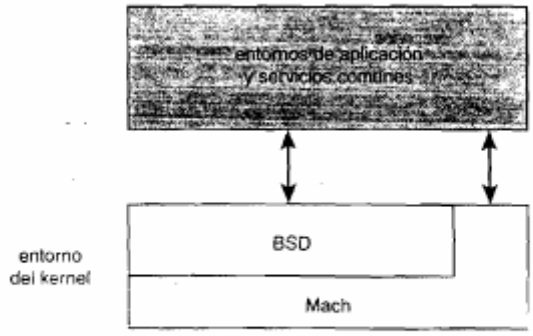

Figura 2.14 Estructura de Mac OS X.

2.8. Máquinas virtuales

La estructura de niveles se lleva a su conclusión lógica con el concepto de la máquina virtual. La idea es **abstraer el hardware de la computadora (CPU, memoria, tarjetas de red, discos...) formando varios entornos de ejecución diferentes, creando así la ilusión de que cada entorno de ejecución está operando en su propia computadora privada.**

Con los mecanismos de planificación de la CPU y las técnicas de memoria virtual, un SO puede crear la ilusión de que un proceso tiene su propio procesador, con su propia memoria (virtual). Normalmente, un proceso utiliza características adicionales, tales como llamadas al sistema o el sistema de archivos, que el hardware básico no proporciona. El método de máquina virtual no da ninguna de esas funcionalidades, sino que **da una interfaz idéntica al hardware básica subyacente. Cada proceso dispone de una copia virtual de la computadora subyacente.**

Existen varias razones para crear una VM, estando todas ellas fundamentalmente relacionadas con el poder compartir el mismo hardware y, a la vez, **operar con entornos de ejecución diferentes concurrentemente**. Durante esta sección, vamos a ver el sistema operativo VM para los sistemas IBM, que constituye un útil caso de estudio; además IBM fue una de las empresas pioneras en este área.

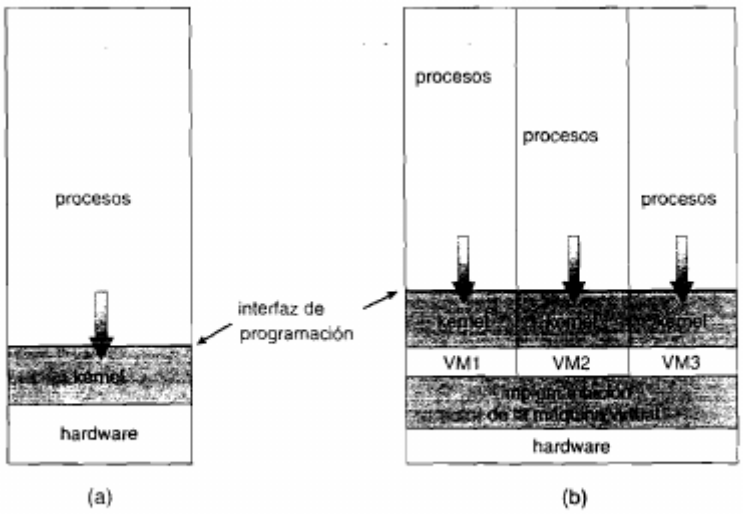

Figura 2.15 Modelos de sistemas. (a) Máquina no virtual. (b) Máquina virtual.

Uno de los principales problemas son los sistemas de disco. Si una máquina física dispone de 3 sistemas de disco, y quiere alojar 7 VM, no puede darle un disco a cada una. No solo porque no dan las mates, incluso si el número de máquinas fueran 3, no se podría porque el sistema requiere una buena porción de disco para proporcionarles memoria virtual a las VM y para los mecanismos de gestión de colas. La solución son discos virtuales (minidiscos en IBM), idénticos en

todo excepto por el tamaño. Obviamente, la suma de los minidiscos no puede superar el espacio físico de discos.

Cuando los usuarios disponen de VM pueden ejecutar cualquier SO o software disponible en la máquina subyacente. En IBM, los usuarios ejecutan normalmente CMS, un SO interactivo monousuario. El software de la máquina virtual se ocupa de multiprogramar las VM sobre una máquina física, sin preocuparse de nada relativo al software de soporte al usuario. Esta arquitectura proporciona una forma muy útil de dividir el problema de diseño de un sistema interactivo multiusuario en dos partes más pequeñas.

2.8.1. Implementación

Es un concepto útil pero difícil de implementar. Es difícil dar un duplicado exacto de la máquina host: recordemos que tenemos dos modos, el kernel y el usuario. **El software de la VM puede ejecutarse modo kernel, dado que es SO; pero la VM en sí sólo puede ejecutarse como modo usuario. Y sin embargo, también tiene que ser capaz de replicar la dicotomía user/kernel.** Por tanto tenemos que crear un modo kernel virtual y modo usuario virtual, que se ejecutan sobre el modo usuario físico.

Las acciones que dan lugar a la transferencia usuario->kernel, en una máquina host (como p.ej. una llamada al sistema o una instrucción privilegiada) también tiene que hacer que se pase usuarioV->kernelV en una VM. Tal transferencia se consigue así: cuando se hace una llamada al sistema por parte de un programa en la VM en modo usuarioV, **se produce una transferencia al monitor de la máquina virtual en la máquina real . Cuando el monitor de la máquina virtual obtiene el control, puede cambiar el contenido de los registros y el contador de programa para que la máquina virtual simule el efecto de la llama a a sistema.** A continuación, puede reiniciar la máquina virtual, que ahora se encontrará en modo kernelV.

Por supuesto la mayor diferencia es el tiempo. Mientras que una E/S real puede tardar 100ms, la virtual puede llevar menos (puesto que se pone en cola) o más (puesto que es interpretada). Además, la CPU se multiprograma entre muchas MV, ralentizándolas entre sí de formas impredecibles. En casos extremos es posible tener que simular todas las intrucciones para proporcionar una verdadera MV. En IBM funciona bien porque las instrucciones normales de la MV se pueden ejecutar por hardware. Sólo instrucciones privilegiadas (E/S) deben simularse y, por tanto, van más lentas.

2.8.2. Beneficios

- Existe una protección completa de varios recursos del sistema: cada VM está completamente aislada de las demás, así que no hay problemas de protección
    - Por otro lado, esto implica que no hay compartición directa de recursos. Se ha intentado solucionar de dos formas, basadas en conceptos ya aplicables sobre recursos físicos, pero implementadas en software:
        - Compartir minidisco, por tanto, comparten archivos.
        - Definir una red de VM por la cual pueden mandar información a través de una red de comunicaciones virtual
- Ideales para la investigación y desarrollo de SO. La potencia del SO hace que su modificación sea peligrosa. Dado que el sistema operativo se ejecuta en modo kemel, un cambio erróneo en un puntero podría dar lugar a un error que destruyera el sistema de archivos completo. Por tanto, es necesario probar cuidadosamente todos los cambios.

- El SO opera la máquina completa. El sistema debe detenerse y queda fuera de uso cuando se hacen y prueban cambios (tiempo de desarrollo del sistema). Este tiempo suele hacerse en horas de poco tráfico para no interrumpir la actividad de usuarios. Una máquina virtual puede eliminar gran parte de este problema. Los programadores de sistemas usan VM para los cambios, en lugar de la máquina física.

2.8.3. Ejemplos

A pesar de las ventajas de las máquinas virtuales, en los años posteriores a su desarrollo recibieron poca atención. Actualmente las máquinas virtuales se están poniendo de nuevo de moda como medio para solucionar problemas de compatibilidad entre sistemas.

En esta sección, exploraremos dos populares máquinas virtuales actuales: VMware y la máquina virtual Java. Como veremos, normalmente estas máquinas virtuales operan por encima de un sistema operativo de cualquiera de los tipos que se han visto con anterioridad. Por tanto, los distintos métodos de diseño de sistemas operativos (en niveles, basado en microkernel, modular y máquina virtual) no son mutuamente excluyentes.

2.8.3.1. VMware

//TODO

2.8.3.2. Máquina Virtual Java

//TODO

Love

Tarea 1.2: Capítulo 1

Capítulo 1: Introducción al kernel Linux

Este capítulo presenta la idea del kernel Linux y el SO Linux, ubicándolos en el contexto histórico de UNIX. Hoy, UNIX es una familia de SO que implementan una API (application program interface) similar, construidos alrededor de ciertas decisiones de diseño compartidas. Pero UNIX también es un SO creado hace más de 40 años. Para entender Linux primero discutiremos el entorno UNIX.

1.1. Historia de UNIX

Tras 40 años UNIX sigue considerándose uno de los sistemas más elegantes y potentes del mundo. Fue creado en 1969 a partir de Multics, un SO multiusuario fallido desarrollado en Bell. Tras la cancelación de Multics los científicos de Bell se vieron sin un SO interactivo funcional y desarrollaron un filesystem que acabaría convirtiéndose en UNIX. Primero se implementó en una máquina PDP7 que tenían ociosa, y luego se porteó a una PDP11. Luego se reescribió en C, lo nunca visto. La primera versión usada fuera de Bell fue V6.

Otras compañías portearon UNIX a otras máquinas, y a su vez incorporaron mejoras, por lo que surgieron variantes que se acabarían juntando en un solo sistema, UNIX system III. La simplicidad de UNIX y el hecho de que se distribuyera con código fuente hizo que cualquiera pudiera añadir mejoras. Algunas de las más relevantes vienen de la Universidad de Berkeley, llamadas BSD. Su licencia laxa hace que aún hoy se sigan desarrollando, como Darwin (MAC OS X), FreeBSD, NetBSD...

En los 80-90 muchas compañías desarrollaron sus propias versiones comerciales, basadas en versiones de AT&T o BSD, que normalmente incluían soporte para su hardware específico. Algunas son Digital's Tru64, Hewlett Packard's HP-UX, IBM's AIX, Sequent's DYNIX/ptx, SGI's IRIX, y Sun's Solaris & SunOS.

El elegante diseño de UNIX y el desarrollo e innovación constante posterior crearon un sistema potente, robusto y estable. Algunas de sus características clave para su éxito son:

- Unix es simple. Muchos sistemas implementan miles de llamadas al sistema y no tienen metas claras en su diseño, mientras que UNIX tiene unos cientos con un diseño básico, straightforward
- En Unix, todo es un archivo. Esto simplifica la gestión de la información con unas pocas llamadas clave: open, read, write, lseek, close.
- El kernel Unix y sus funcionalidades del sistema están escritas en C. Esto le da mucha portabilidad a diversas estructuras hardware y lo hace accesible a muchos más desarrolladores.
- Unix tiene una rápida creación de procesos, y sólo con la llamada al sistema fork
- Unix da primitivas de comunicación interprocesos simples y robustas que, cuando se combinan con la rápida creación de procesos mencionada antes, nos dejan programas que tienen una sola función, pero la hacen bien. Estos programas de un sólo uso pueden combinarse para solucionar tareas de mayor complejidad. Los sistemas Unix exhiben capas bien definidas, con una clara separación entre políticas y mecanismos.

Hoy, Unix es un sistema moderno que soporta multitasking, multithreading, memoria virtual, paginación, bibliotecas compartidas, redes TCP/IP... Muchas variantes UNIX encompasan cientos de procesadores, mientras que otras corren en pequeños sistemas embebidos. Unix ya no es un pequeño proyecto de investigación, pero sigue beneficiándose de los avances en los SO; y a la vez sigue siendo un sistema práctico de uso general.

Unix debe su éxito a su elegancia y simpleza de diseño. Su fueza hoy deriva de las decisiones de diseño tomadas en su día por Dennis Ritchie, Ken Thompson, y otros desarrolladores en Bell: decisiones que hoy permiten a Unix evolucionar sin cambiar.

1.2. Presentando Linux

Linux Torvalds desarrolló la primera versión de Linux en 1991 como un SO para ordenadores con el Intel 80386, un procesador nuevo y avanzado en esa época. Linux era universitario y echaba en falta un sistema Unix potente y gratuito. Microsoft DOS reinaba en PC y Minix, una distribución low-cost de Unix, no podía modificarse por problemas de licencia, además de tener ciertas decisiones de diseño con las que Linus no estaba de acuerdo. Así que Linus comenzó a desarrollar su propio SO, inicialmente un emulador de terminal que usó para conectarse a sistemas Unix mayores en su facultad. Fue desarrollando y para final de curso tuvo un sistema Unix, inmaduro pero funcional, que subió a internet.

Tuvo éxito por su licencia, que permitió a desarrolladores de todo el mundo añadir, modificar y mejorar Linux hasta convertirlo en un gran proyecto colaborativo.

Linux es un sistema basado en Unix, pero no es Unix. Toma muchas ideas prestadas e implementa la API de Unix (definida por POSIX y Single Unix Specification), pero no es descendiente de Unix en el aspecto del código. Se han tomado decisiones que se desviaban de Unix allá donde se ha considerado necesario, pero generalmente no se ha olvidado de las metas de diseño de Unix.

Uno de los aspectos más interesantes de Linux es que no es un producto comercial, sino un proyecto colaborativo desarrollado en internet. Linus es el creador y el encargado de mantener el kernel, pero su desarrollo lo lleva un grupo de individuos en internet. Cualquiera puede contribuir: es open source licenciado con GPL. Esto significa que eres libre de descargarte el código fuente y modificarlo, pero debes distribuirlo bajo la misma licencia.

Linux es muchas cosas, pero en el fondo es un kernel, con bibliotecas C, toolchain y utilidades del sistema básicas como un login y la shell. Podríamos hablar de interfaces gráficas y mil cosas más, pero a partir de ahora cuando hablemos de Linux, a no ser que se diga lo contrario, hablaremos del kernel de Linux.

1.3. Vistazo a los sistemas operativos y kernels

Por su rápido crecimiento y por cuestionables decisiones de diseño en SO comerciales modernos, la definición de lo que es un SO no es universal. Aquí consideramos un SO las partes del sistema responsables del uso básico y la administración, lo cual incluye: el kernel, los drivers de dispositivos, boot loader, shell)u otra interfaz de usuario para comandos), y utilidades básicas de archivos y sistema. Por otro lado, definimos sistema como el SO y todas las aplicaciones que corren en él.

El tema de este libro es el kernel. Si la interfaz de usuario es el exterior del sistema, el kernel es el núcleo. Es el software que provee servicios básicos para todas las demás partes del sistema, gestiona el hardware y distribuye los recursos. Componentes típicos del kernel serían:

- handlers de interrupciones para manejar las solicitudes de interrupción,
- un scheduler para gestionar el tiempo de CPU entre los procesos,
- un sistema de gestión de memoria para manejar las direcciones de los procesos,
- y servicios del sistema como redes o comunicación interproceso.

En sistemas modernos con unidades de gestión de memoria protegidas, el kernel reside en una zona superior a las aplicaciones de usuario. Esto suele incluir un espacio de direcciones reservado y acceso libre al hardware. El estado del sistema y la memoria reservada se llama espacio del kernel. Por otro lado, las aplicaciones de usuario residen en el espacio de usuario, y ven una fracción de los recursos de la máquina, lo que permita el kernel. Cuando corremos un proceso kernel, estamos usando el espacio de kernel y estamos en modo kernel; lo respectivo sucede con las aplicaciones de usuario.

Las aplicaciones en el sistema se comunican con el kernel mediante llamadas al sistema. Una aplicación llama funciones en una biblioteca, y ésta usará la interfaz de llamadas al sistema para llamar al kernel.

//TODO

1.4. Linux vs. Kernels clásicos Unix

1.5. Versiones del kernel Linux

Los kernels de Linux pueden ser dos tipos: estables o desarrollo. Los estables son releases de producción aptos para despliegue general, y normalmente proveen arreglos de bugs o nuevos drivers. Los kernel desarrollo sufren cambios rápidamente, según se experimentan soluciones.

Linux kernels distinguish between stable and development kernels with a simple naming scheme (see Figure 1.2). Three or four numbers, delineated with a dot, represent Linux kernel versions. The first value is the major release, the second is the minor release, and the third is the revision. An optional fourth value is the stable version. The minor release also determines whether the kernel is a stable or development kernel; an even number is stable, whereas an odd number is development. For example, the kernel version 2.6.30.1 designates a stable kernel. This kernel has a major version of two, a minor version of six, a revision of 30, and a stable version of one. The first two values describe the "kernel series"—in this case, the 2.6 kernel series.

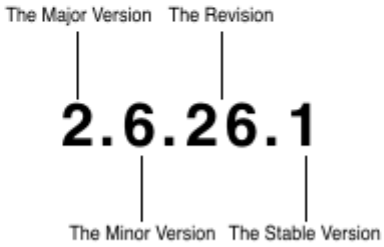

Figure 1.2 Kernel version naming convention.

Los kernels desarrollo tienen varias fases. Al principio los desarrolladores trabajan nuevas funcionalidades, lo que es un caos. Con el tiempo, el kernel va madurando y Linus declara un feature freeze, lo que significa que no se aceptan más funcionalidades. Se trabaja sobre las que se han ido desarrollando. Cuando llegan a cierto punto, Linus declara un code freeze: ahora sólo se pueden arreglar bugs, no se puede añadir nada nuevo. Poco después, si todo va bien, saldrá la versión estable.

Así se desarrolló hasta 2004, hasta que en el Kernel Developers Summit (un evento con invitados selectos) se decidió que la versión 2.6 se iba a prolongar, porque era una versión estable, bien recibida y madura; y era preferible mantenerla y mejorarla, a desestabilizarla con nuevas funcionalidades.

1.6. La comunidad de desarrollo del kernel Linux

Si desarrollas código para el kernel Linux, pasarás a formar parte de la comunidad global de desarrollo Linux. El foro principal de esta comunidad es la Linux Kernel Mailing List (lkml) - es una lista con un tráfico de cientos de mensajes al día, por gente que no está para bromas (entre ellos Linus y los desarrolladores del kernel). Pero es necesaria porque es el lugar donde encontrar testers, peer-reviewers, y preguntar.

Mauerer

Tarea 1.2: Secciones 7.1 y 7.2 -- "The database información is provided ..." NO

Tarea 1.2: Secciones 2.3.2 y 2.8.2

7.1. Primer vistazo

7.2. Usando módulos

7.2.1. Quita y pon

7.2.2. Dependencias

7.2.3. Querying información del módulo

7.2.4. Carga automática