9 . Modo dual (usuario y *kernel*)

El código del propio sistema operativo, es decir, su conjunto de instrucciones para gestionar recursos y proporcionar servicios, se encuentra codificado en lenguaje máquina y cargado en memoria principal. Dado que el sistema operativo, los procesos y usuarios comparten los recursos hardware y software del sistema informático, necesitamos asegurar que un ***error*** *que se produzca en un programa de usuario s****ólo genere problemas en el programa que se estuviera ejecutando****, y no en el control del hardware o incluso en otros usuarios y programas* **(independencia de errores)**.

Imaginemos que el usuario a través de un programa de aplicación, pudiera hacer cambios en tiempo real de las rutinas o instrucciones máquina que rigen el funcionamiento del sistema operativo y que interaccionan con el hardware, imaginemos que pudiera cambiar el manejador de interrupciones, la forma en que se produce comunicación con un dispositivo de E/S; imaginemos que pudiera cambiar la codificación del sistema operativo o acceder a ella para utilizarla a su manera y gusto. Si esto sucediera se podría hacer un **uso indebido del sistema**, en última instancia del hardware, produciendo comportamientos inesperados y pudiendo corromper el funcionamiento y gestión de los recursos, programas y usuarios.

Para evitar esto los diseñadores han buscado una solución basada en un modo dual de operación:

- El **modo núcleo, *kernel*, supervisor o privilegiado** (no tiene nada que ver con el *root*)**:** Este modo está asociado al procesamiento de instrucciones y rutinas del sistema operativo. La gestión de recursos del sistema y los servicios que éste ofrece están accesibles solo a través de lo que se llaman **llamadas al sistema.** Las llamadas al sistema son rutinas o funciones nativas a nivel de núcleo, es decir, situadas en la parte de la memoria principal reservada al sistema operativo.

  Por tanto, el núcleo es la parte del sistema operativo que engloba las funciones y servicios más importantes del sistema y que deben protegerse, y suele residir en lenguaje máquina en memoria principal de forma continua, mientras que el resto del sistema operativo se carga cuando es necesario (si el sistema tienen un kernel modular – se estudia posteriormente –).

- El **modo usuario:** Este modo está asociado al procesamiento de instrucciones y rutinas de los programas de usuario, es decir, ejecuta rutinas cuyo código máquina y datos está situado en la parte de memoria principal reservada a dichos programas. Un programa de usuario no puede acceder directamente a posiciones de memoria del núcleo del sistema operativo, debe hacerlo a través de la invocación de llamadas al sistema.

De forma general, **¿cuándo se ejecuta o se accede al código del núcleo?**

- Cuando una aplicación ejecuta una **llamada al sistema**, ya sea a través de una API, desde el *Shell* o de cualquier otra manera.
- Cuando una aplicación provoca una **excepción** (ya que hay que ejecutar las rutinas del núcleo necesarias para tratarla)**,** que es una situación de error detectada por la CPU mientras ejecutaba una instrucción, que requiere tratamiento por parte del SO.

  Por ejemplo, división por cero, acceso a direcciones de memoria no permitido, violación de permisos, etc. Las excepciones son muy parecidas a las interrupciones, en este caso las excepciones suelen ser de tipo software.
- Cuando un dispositivo provoca una **interrupción** (a través del modulo de E/S), ya que hay que ejecutar las rutinas del núcleo necesarias para tratar esa interrupción.

La Tabla 3.7 lista las funciones básicas que normalmente se encuentran dentro del núcleo del sistema operativo:

Tabla 3.7. Funciones típicas de un núcleo de sistema operativo.

Gestión de procesos

- Creación y terminación de procesos
- Planificación y activación de procesos
- Intercambio de procesos
- Sincronización de procesos y soporte para comunicación entre procesos
- Gestión de los bloques de control de proceso

Gestión memoria

- Reserva de espacio direcciones para los procesos
- *Swapping*
- Gestión de páginas y segmentos

Gestión E/S

- Gestión de *buffers*
- Reserva de canales de E/S y de dispositivos para los procesos

Funciones de soporte

- Gestión de interrupciones
- Auditoría
- Monitorización

Dicho esto, **¿cómo se cambia de modo?** Existe típicamente un **bit en la palabra de estado** de programa (PSW) que indica el modo de ejecución, este bit se cambia como respuesta a determinados eventos, como pueden ser la ocurrencia de una interrupción, una excepción o de una llamada al sistema.

De esta manera el hardware detecta los errores de violación de los modos y es el sistema operativo el que se encarga de tratarlos, del tal forma que si desde un programa de usuario se hace un intento de ejecutar una instrucción privilegiada del núcleo, el hardware envía una excepción al sistema operativo. La excepción transfiere el control al sistema operativo a través del vector de interrupción, igual que cuando se produce una interrupción. Entonces el sistema operativo proporciona un mensaje de error apropiado, el cual puede volcarse en memoria. Normalmente, el volcado de memoria se escribe en un archivo con el fin de que el usuario o programador puedan examinarlo y quizá corregir y reiniciar el programa.

La Figura 1.8 muestra gráficamente un cambio de modo ante una llamada al sistema. Como dato curioso, en cualquier S.O. moderno, como lo es Gnu/Linux, el procesador alterna entre ambos modos al menos unas cuantas miles de veces por segundo.

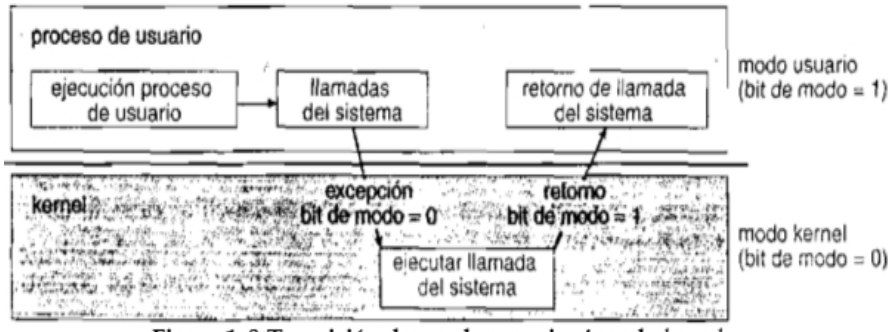


Figura 1.8 Transición de¡ modo usuario a! modo *kernel*.

El SO configura periódicamente una interrupción de reloj para evitar perder el control y cambiar de modo, de forma que un usuario no pueda acaparar todos los recursos del sistema.

10. Llamadas al sistema

Una llamada al sistema es utilizada por una aplicación (programa de usuario) para **solicitarle un servicio al sistema operativo.** Las llamadas al sistema por tanto **proporcionan un interfaz entre un programa y el sistema operativo** para poder invocar los servicios que éste ofrece y gestionar sus recursos, ya que como se ha comentado antes con los modos duales, el usuario no puede acceder o no tiene privilegios directos sobre los recursos que gestiona el sistema operativo.

Antes de ver cómo pone un sistema operativo a nuestra disposición las llamadas al sistema, vamos a ver un ejemplo para ilustrar cómo se usan esas llamadas. Suponga que deseamos escribir un programa sencillo para leer datos de un archivo y copiarlos en otro archivo. El primer dato de entrada que el programa va a necesitar son los nombres de los dos archivos, el de entrada y el de salida - ***int copy_file(char* f_org,char* f_dest)*** - Para hacer esta operación se requerirá una secuencia de llamadas al sistema:

1. Una vez que se han obtenido los nombres de los dos archivos, el programa debe abrir el archivo de entrada (llamada al sistema) y crear el archivo de salida (llamada al sistema).
2. Cuando el programa intenta abrir el archivo de entrada, puede encontrarse con que no existe ningún archivo con ese nombre, o que está protegido contra accesos. En estos casos, el programa debe devolver un error y terminar de forma anormal controlada.
3. Si el archivo de entrada existe, entonces se debe crear un nuevo archivo de salida (llamada al sistema). En este caso, podemos encontrarnos con que ya existe un archivo de salida con el mismo nombre; esta situación puede hacer que el programa termine o podemos borrar el archivo existente (llamada al sistema) y crear otro (llamada al sistema).
4. Una vez que ambos archivos están definidos, hay que ejecutar un bucle que lea del archivo de entrada (llamada al sistema) y escriba en el archivo de salida (llamada al sistema). Cada lectura y escritura debe devolver información de estado relativa a las distintas condiciones posibles de error: el programa puede encontrarse con que ha llegado al final del archivo, o incluso con que se ha producido un fallo de hardware (espacio de disco insuficiente, sector dañado, etc.)
5. Finalmente, después de que se ha copiado el archivo completo, el programa cierra ambos archivos (llamada al sistema) y termina normalmente.

Lo que este ejemplo quiere hacerle ver es que incluso los programas más sencillos hacen un uso intensivo del sistema operativo y de servicios y rutinas implementadas en su núcleo. Normalmente, los sistemas ejecutan **miles de llamadas al sistema por segundo**, y son transparentes al programador y al usuario final.

Los programadores pueden usar las **llamadas al sistema de dos formas:**

1. Normalmente, los desarrolladores de aplicaciones diseñan e implementan sus programas utilizando una **API (*application programming interface*, interfaz de programación de aplicaciones)**. La API especifica un conjunto de funciones que el programador de aplicaciones puede usar, indicándose los parámetros que hay que pasar a cada función y los valores de retorno que el programador debe esperar, no se tiene por qué saber nada acerca de cómo se implementa dicha función invocada o qué es lo que ocurre durante su ejecución. Por tanto, la API oculta al programador la mayor parte de los detalles de la interfaz del sistema operativo y de las llamadas nativas disponibles.

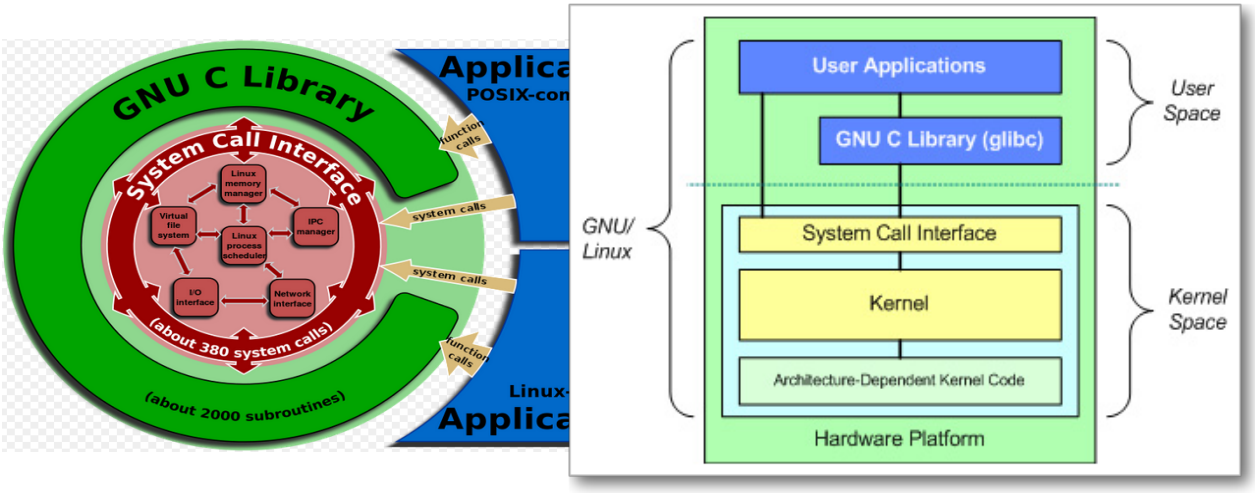
   Dos de las API más usuales disponibles para programadores de aplicaciones son la API Win32 para sistemas Windows y la API de la biblioteca estándar de C, *glibc*, para sistemas basados en POSIX (prácticamente todas las versiones de Gnu/Linux y Mac OS).

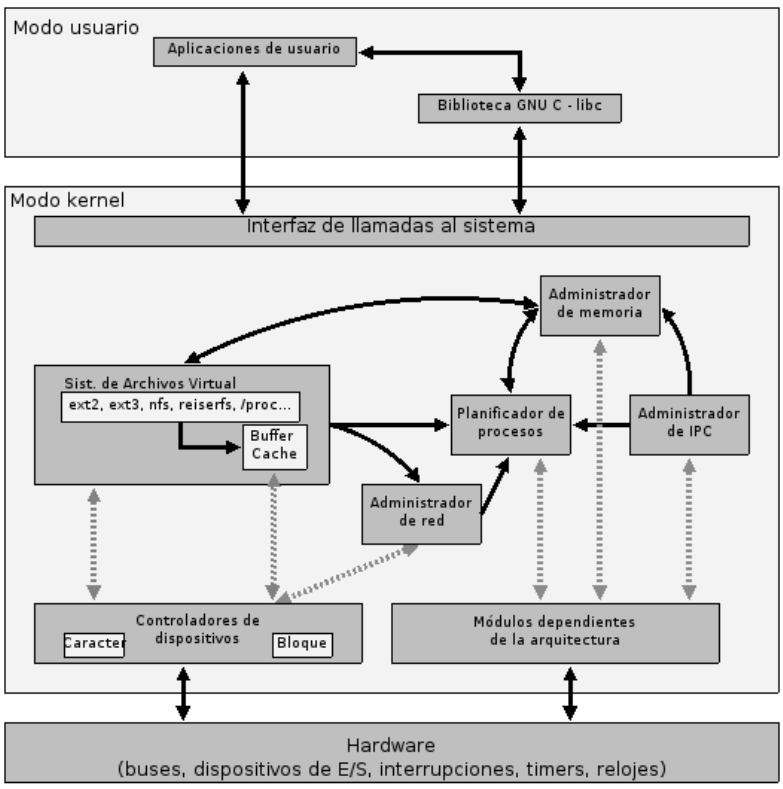
   Las funciones que conforman una API de alto nivel invocan a las llamadas al sistema internamente, realmente actúan como ***wrapers*** o envolventes de **funciones nativas del**

núcleo. Es decir, la propia función invocada no es la llamada al sistema en si, sino que por debajo se encuentra una llamada o conjunto de llamadas al sistema nativas. Por ejemplo, la función *CreateProcess()* de Win32 (crea un nuevo proceso), lo que hace realmente es invocar la llamada nativa al sistema *NTCreateProcess()* del *kernel* de Windows. Lo mismo pasa con Gnu/Linux, pero los nombres de las llamadas son diferentes.

2. Acceso **mediante la interfaz ofrecida por el núcleo del sistema operativo**, definida a nivel de lenguaje **ensamblador**. Esto es una manera más compleja de usar las llamadas al sistemas, ya que el programador directamente debería usar lenguaje ensamblador para invocarlas. Depende directamente del hardware sobre el cual se está ejecutando el sistema operativo: registros del procesador, cómo se cambia de modo, cómo se salta del código de usuario al código del núcleo (*jump*, *call*, *trap*, *int*, *sysenter* ...), etc.

Las siguientes figuras muestran gráficamente dónde se sitúan las llamadas al sistema y a qué nivel de abstracción:





En la Figura 2.6 se muestra un ejemplo de una llamada en C para escribir por la salida estándar del sistema. Suponga un programa en C que invoque la sentencia *printf()*. La biblioteca de C intercepta esta llamada e invoca la/s llamada/s nativa/s al núcleo que se necesiten, en este caso solo hace falta invocar a la llamada nativa del sistema *write(),* pero podrían hacer falta más. Posteriormente, la biblioteca de C toma el valor que devuelve *write*() y lo pasa de nuevo al programa de usuario.

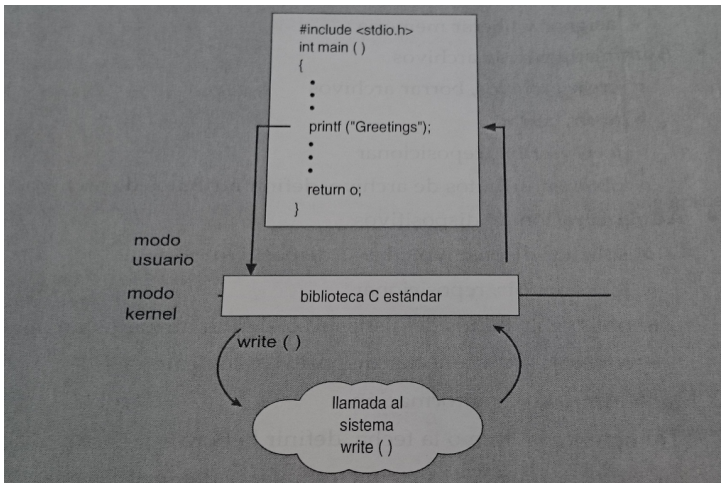


Figura 2.6 *Tratamiento de la instrucción* `write()` *en la biblioteca C .*

Cuando se realiza una llamada al sistema puede ser necesario **pasar parámetros** al núcleo del sistema operativo. De manera general se emplean tres métodos:

1. El más sencillo de ellos consiste en pasar los parámetros a una serie de registros del procesador.
2. En algunos casos, puede haber más parámetros que registros disponibles. En estos casos, generalmente, los parámetros se almacenan en un bloque o tabla en memoria, y la dirección del bloque se pasa como parámetro a un registro.
3. Otra alternativa sería que el programa de usuario colocase los parámetros en la pila de memoria principal reservada al usuario, y el sistema operativo se encargará de extraer de la pila esos parámetros a partir de una dirección de memoria proporcionada. La mayoría de los sistemas operativos prefieren el método del bloque o el de la pila, porque no limitan el número o la longitud de los parámetros que se quieren pasar.

Habitualmente, cada **llamada al sistema tiene asociado un número** y la interfaz de llamadas al sistema mantiene una **tabla indexada en la memoria** con dichos números. Usando esa tabla, la interfaz de llamadas al sistema invoca la llamada necesaria del ***kernel*** del sistema operativo y devuelve el **estado de la ejecución de la llamada al sistema + los posibles valores de retorno**. Esto se hace a través de funciones especiales denominadas de tipo ***trap*** que el programador puede invocar (se comentan en la siguiente sección).

Teniendo en cuenta lo anterior, **¿cómo conseguimos la portabilidad entre diferentes versiones del SO?**

Una llamada a sistema no se identifica con una dirección, sino con un identificador (un número) que usamos para indexar la tabla de llamadas a sistema (que se debe conservar constante entre versiones). Por tanto, la tabla de llamadas a sistema es la ofrece compatibilidad entre diferentes versiones del SO.

Las llamadas al sistema pueden agruparse en 5 categorías. Si observa, prácticamente es lo mismo que las funciones básicas que se encuentran dentro del núcleo del sistema operativo y que

mostraron en secciones anteriores. Dependiendo de la literatura que consulte esta división o categorización puede varias levemente:

1. **Control de procesos:**

   Crear procesos, terminar procesos, definir u obtener atributos de un proceso, esperar o señalizar un suceso, asignar o liberar memoria.

2. **Manipulación de archivos:**

   Crear o borrar, abrir o cerrar, leer o escribir, reposicionar, definir u obtener atributos de archivo.

3. **Manipulación de dispositivos:**

   Solicitar dispositivo, liberar dispositivo, leer, escribir o reposicionar, obtener o definir atributos de dispositivo, conectar y desconectar.

4. **Mantenimiento de información:**

   Obtener o definir la hora y la fecha, obtener o establecer datos del sistema (monitorización), obtener atributos de procesos, archivos o dispositivos, establecer atributos de procesos, archivos o dispositivos.

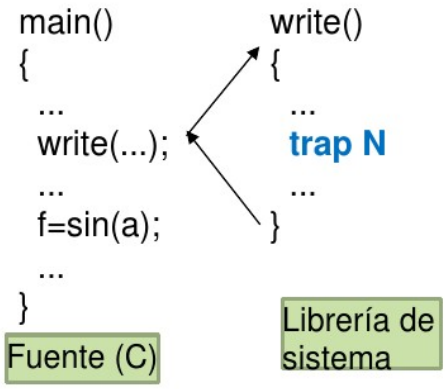
5. **Comunicaciones:**

   Crear o eliminar conexiones de comunicación, enviar o recibir mensajes, transferir información de estado, conectar o desconectar dispositivos remotos.

10.1 Funciones de tipo trap para acceso al núcleo

Una vez presentado el concepto de llamada al sistema, a continuación se muestra qué ocurre cuando se produce una invocación y cómo realizarla. Se mostrará el proceso general mediante el uso de API:

1. Cuando se produce una llamada al sistema a partir de una API, los **parámetros asociados** a la misma se cargarían en la **pila del proceso a nivel de usuario,** o en **registros**, dependiendo del sistema**.**
2. Posteriormente, en la implementación interna de la función de la API invocada, se ejecuta una función especial que ofrece el interfaz de llamadas al sistema del sistema operativo, denominada de tipo ***trap*** (cambiará de nombre según el sistema). Por tanto, desde el punto de vista del programador, *trap* sería invocada a más bajo nivel (aún no en ensamblador), ya que estas funciones *trap* ya no forman parte de la API como tal, sino que son las funciones de la propia API las que la invocan internamente.



3. La función de tipo ***trap*** se encargará de obtener los parámetros almacenados y de ejecutar una **interrupción especial** que conlleva **un cambio de modo**, de modo usuario a modo núcleo supervisor (**NO** tiene nada que ver con el superusario ***root***).
4. Siempre que se realiza una llamada al sistema hay que **guardar el estado** de ejecución actual y el valor del contador de programa o PC, ya que se va a saltar a una zona de memoria donde se encontrará alojada la llamada del sistema invocada (y después no se sabría volver al proceso que se estaba ejecutando). La interrupción especial invocada por la función de tipo ***trap*** se encargará de llamar a otras subrutinas del núcleo para hacer estas operaciones de salvado, justo **antes de empezar a ejecutar la llamada al sistema** concreta a partir de su número de identificación.
5. Posteriormente al salvado se **examina la rutina nativa invocada y sus parámetros**, y se busca en una tabla o vector de rutinas si existe (número de identificación), además de la dirección del lugar del núcleo donde se encuentra para proceder a ejecutarla.
6. Tras identificar que existe una rutina para la llamada realizada y que los parámetros son correctos (se recogen de la pila a nivel de usuario o están almacenados en registros, según el sistema), **se procede a ejecutarla.**
7. Una vez ejecutada la llamada al sistema se invocan otras subrutinas denominadas de tipo **RETURN FROM TRAP**, para devolver el **código de estado de la llamada al sistema y los posibles parámetros de retorno** que pudiera devolver la misma. Se regresa, por tanto, el control a la función de biblioteca (por ejemplo en C), **pasándose a modo usuario**.
8. Cuando finalizan esas acciones, la rutina de biblioteca **a nivel de usuario descarga la pila** y comprueba el resultado de la ejecución de la petición al sistema, de forma que el programa de usuario continúe su ejecución.

Tenga en cuenta que el esquema proporcionado es genérico, y que no se ha tenido en cuenta que la ejecución de la llamada al sistema se pueda interrumpir. Dependiendo del sistema y su configuración o políticas del núcleo, una excepción, una interrupción o una llamada al sistema se podrían ver interrumpidas por otras de mayor prioridad. Si el sistema admite ejecutar interrupciones y procesos de mayor prioridad, debe tener implementado en su núcleo determinadas instrucciones y rutinas que lo contemplen.

Por otro lado, durante la ejecución de la llamada al sistema el proceso llamante ha podido cambiar de estado (se estudiarán los estados de los procesos), o quizás haya señales que atender, por lo que si el planificador lo considera oportuno, al termino de la llamada al sistema se podría pasar a otro proceso diferente.

Por último, si se programase a nivel ensamblador, esas llamadas a las diferentes subrutinas que tienen lugar durante el proceso de invocación de una llamada al sistema se tendrán que hacer cuidadosamente, siguiendo el orden necesario para que todo lo comentado se realice correctamente. Este tipo de programación la suelen hacer los diseñadores y programadores del sistema operativo, siendo poco frecuente en el programador habitual. En GNU/Linux, los ficheros programados en **lenguaje ensamblador** tienen la **extensión *.S***.

10.2 Llamadas al sistema en GNU/Linux

A continuación se muestra como se lleva a cabo el mecanismo mediante el cual GNU/Linux implementa las **llamadas al sistema en una arquitectura x86 (32 bits)**. Tenga en cuenta que dependiendo de la arquitectura y dependiendo de la versión del sistema operativo y su kernel, las siguientes indicaciones pueden variar. Por tanto, para saber cómo se producen las llamadas al sistema siempre tiene que acudir a información específica del sistema y versión que desee estudiar.

A nivel de API, las llamadas al sistema en GNU/Linux se encuentran descritas en la sección 2 del manual. Usando el comando *prompt>* ***man man*** puede ver qué se ofrece en cada sección.

Usando *prompt>* ***man 2 <nombre_de_la_funcion>*** puede obtener información de la función concreta que especifique.

A nivel de lenguaje ensamblador, para la versión 2.6 del kernel de Linux y para la arquitectura x86, se utiliza la interrupción especial ***int 0x80*** como manera de acceder a una llamada al sistema.

Para la arquitectura y versión de núcleo comentada, en el fichero ***include/asm-i386/unistd.h*** (dependiendo de la versión del kernel la localización, nombres de directorios, ficheros y funciones pueden cambiar) aparecen listadas las llamadas al sistema que ofrece Linux con su correspondiente número, por ejemplo **__NR_close**, que corresponde a la llamada nativa al sistema ***sys_close()***, que es la función ***close()*** a nivel de API.

int 0x80*:* Interrupción especial que iniciará el proceso de acceso a la llamada al sistema que se vaya a invocar:

- Esta interrupción se invoca a partir de la rutina ***__init trap_init()***, invocada a su vez por una rutina llamada ***start_kernel(),*** que proviene a su vez de una función a nivel de API.
- La función ***__init trap_init()*** se encuentra en el fichero ***arch/x86/kernel/traps.c***
- El número de la interrupción software ***int 0x80*** está definido por la constante **SYSCALL_VECTOR**, que se encuentra definida en el fichero ***arch/x86/include/asm/irq_vectors.h***

Dicho esto, en el siguiente código a partir de la función ***__init trap_init(),*** se establece el método de entrada al núcleo mediante la función ***set_system_gate()***. La función ***set_system_gate()*** invoca a la interrupción ***int 0x80,*** que producirá un salto a la zona de memoria donde se encuentra la función ***system_call()***, situada en el fichero en ensamblador ***arch/x86/kernel/entry_32.S***.

En el **proceso de salto** el procesador pasa de **modo usuario a modo núcleo**, y se cambia el puntero de pila para que apunte a la pila del núcleo del proceso donde está la función ***system_call().*** Observe que la función ***__init trap_init()*** está en modo usuario antes de invocar a ***set_system_gate()***.

arch/i386/kernel/traps.c [995-996,1009-1017,1032,1037-1040]

```
 995 void __init trap_init(void)
 996 {
...
1009         set_trap_gate(0,&divide_error);
1010         set_intr_gate(1,&debug);
1011         set_intr_gate(2,&nmi);
1012         set_system_intr_gate(3, &int3); /* int3-5 can be called from all */
1013         set_system_gate(4,&overflow);
1014         set_system_gate(5,&bounds);
1015         set_trap_gate(6,&invalid_op);
1016         set_trap_gate(7,&device_not_available);
1017         set_task_gate(8,GDT_ENTRY_DOUBLEFAULT_TSS);
...
1032         set_system_gate(SYSCALL_VECTOR,&system_call);
...
1037         cpu_init();
1038
1039         trap_init_hook();
1040 }
```

Ahora en modo núcleo los pasos son los que se exponen a continuación. Para intentar dar una mejor comprensión del complejo proceso de llamadas al sistema y aligerar la lectura, se omiten algunas instrucciones, ejecución de subrutinas y otras comprobaciones:

1. La función ***system_call()*** guarda en su pila el código o i**dentificador de la llamada al sistema** que se quiere invocar. Lo hace en posición donde está situado el registro ***%eax.***

2. Acto seguido ***system_call()*** también guarda todos los **registros del procesador** con la macro **SAVE_ALL** (para la futura restauración del contexto del proceso de usuario interrumpido por ***int 0x80***), situada en el archivo en ensamblador ***arch/x86/kernel/entry_32.S.*** Y también se guardan en la pila los **registros de la pila de la función interrumpida a nivel de usuario** (parámetros de entrada y salida, dirección de retorno, etc).

3. Se comprueba que el número de llamada pedido es válido mediante la estructura o tabla ***sys_call_table***. Si no es válido, se ejecuta el código de la subrutina **syscall_badsys**, situado en el archivo en ensamblador ***arch/x86/kernel/entry_32.S***, que devuelve el código de error **ENOSYS (**para ponerlo posteriormente en ***errno***), y se salta al código código de la subrutina ***resume_userspace***, también situado en el archivo en ensamblador ***arch/x86/kernel/entry_32.S.***

4. Si la llamada se puede realizar **se invoca a dicha llamada** (YA POR FIN!!!) y se guarda el **valor de retorno** en la posición de la pila del núcleo donde está situado el registro ***%eax***. Dicho registro tenía hasta ese momento el número identificativo de la llamada al sistema a invocar. En la arquitectura x86, Linux devuelve el código de retorno de la llamada al sistema en el registro ***%eax***.

   La llamada al sistema nativa que siempre tiene esta forma de prototipo: ***sys_nombre-de-la-funcion(parámetros).*** Los parámetros se recogen de la pila de la función ***system_call().***

   La llamada al sistema puede ser una petición de algún fichero a un dispositivos periférico de E/S, una impresión por pantalla, la obtención de la hora local, etc.

5. Al finalizar la ejecución de la llamada al sistema se ejecuta el código que se encuentra en la subrutina ***syscall_exit***, situada en el archivo ***arch/x86/kernel/entry_32.S***. Se realizan una serie de comprobaciones del estado del proceso a nivel de usuario que se interrumpió, y si no hay nada más prioritario se procederá a su restauración

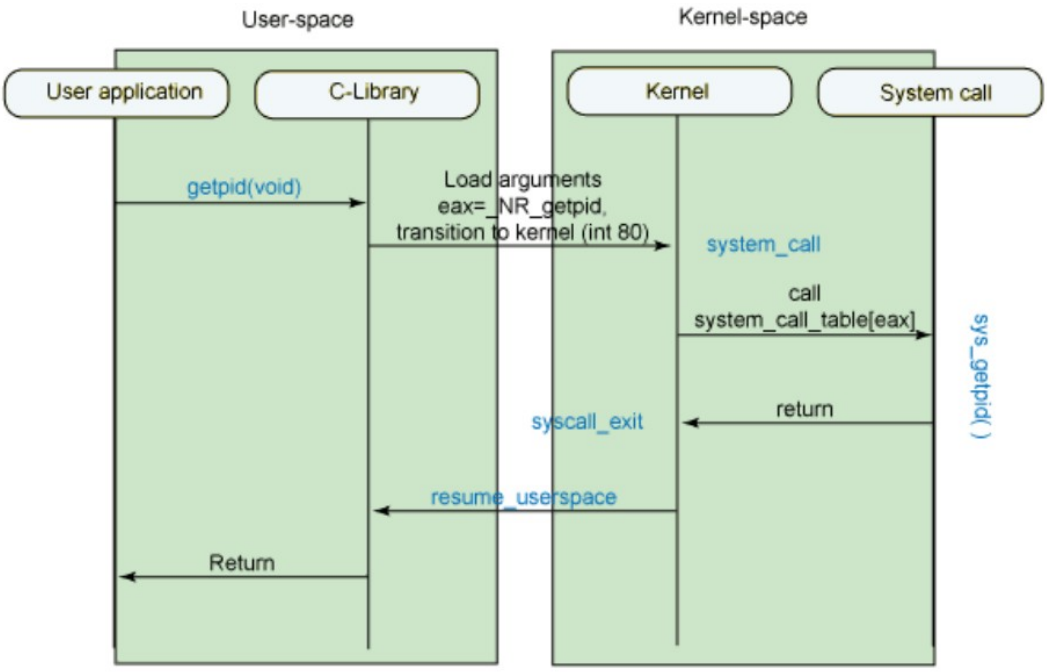
6. Para restaurar el proceso de usuario se ejecuta la macro **RESTORE_ALL,** situada en el archivo ***arch/x86/kernel/entry_32.S***. Esta macro restaura los registros almacenados con **SAVE_ALL.**

7. A continuación se ejecuta la instrucción de retorno de interrupción ***iret***. y se vuelve de la interrupción ***int 0x80***, restaurando los valores necesarios en la pila a nivel de usuario (valor devuelto por la función incluido, que está almacenado en ***%eax***), y se cambia a modo usuario.

8. Si la llamada es exitosa la biblioteca a nivel de usuario (***glibc***) obtiene el resultado y seguirá la ejecución del programa de usuario por donde iba.

9. Cuando la llamada no ha tenido éxito, el valor devuelto ***%eax*** en es negativo (-***1***). Si es negativo, se copia el tipo de error sobre una variable global llamada ***errno*** **y se devuelve** -***1*** a la pila de usuario como valor de retorno de la función. La biblioteca a nivel de usuario podría comprobar el valor guardado en la variable global ***errno***, que contiene el código de error de la última llamada que falló. Una llamada que se realice con éxito no modifica ***errno.***

En la siguiente figura se muestra un resumen gráfico del proceso de una llamada al sistemas desde una API:



11. Diseño del sistema operativo

En esta sección se comentan brevemente las estructuras más importantes que se han probado en los sistemas operativos con respecto a la organización del *kernel*, aunque existen más. La estructuración del sistema difiere de unas metodologías a otras dependiendo del número de capas y de las funcionalidades o servicios dispuestos en cada una de ellas, hasta llegar al kernel y su ejecución en modo privilegiado en el espacio de memoria no disponible para el usuario.

11.1 Sistemas Monolíticos

En este caso el sistema operativo no se organiza en varias capas, sino que se escribe como una **colección de procedimientos enlazados entre sí en un solo programa binario ejecutable** extenso, con todos los elementos compartiendo el mismo espacio de direcciones (en el siguiente tema se explica más detalladamente el concepto de proceso y su ocupación y estructuración como ente en memoria). Por tanto un sólo proceso privilegiado (justamente el sistema operativo) es el que opera en modo supervisor, dentro del cual se encuentran todas las rutinas para las diversas tareas que realiza el sistema operativo.

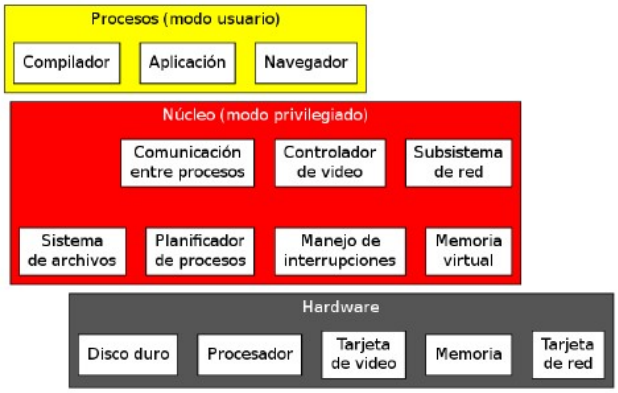


Figura 1.9: Esquematización de los componentes en un sistema monolítico.

Cuando se utiliza esta técnica, en términos de ocultamiento de información, cada procedimiento o rutina en el sistema tiene la l**ibertad de llamar a cualquier otra (son visibles)**, si ésta proporciona cierto cómputo útil que la primera necesita. Al tener miles de procedimientos que se pueden llamar entre sí sin restricción, con frecuencia se produce un **sistema poco manejable y difícil de comprender**. Un ejemplo de ello es el sistema operativo MS-DOS y la primera versión de *Unix*.

La principal **ventaja** de diseñar un sistema siguiendo un esquema monolítico es la **simplificación** de una gran cantidad de mecanismos de comunicación, que lleva a una mayor **velocidad de ejecución**.

El **problema** que plantean este tipo de sistemas radica en lo complicado que es **modificar el sistema operativo** para añadir nuevas funcionalidades y servicios, siendo complicado su mantenimiento. Añadir una nueva característica al sistema operativo implica la modificación de un gran programa, compuesto por miles de líneas de código fuente y funciones.

Esto es todo lo contrario a lo que presentaban los sistemas estructurados en múltiples capas mostrados al inicio de este tema, mucho más modulares y la con ocultación de información de unas capas a otras, facilitando enormemente la depuración y verificación del sistema, puesto que las capas se pueden ir construyendo y depurando por separado.

11.2 Sistemas de Microkernels

Como se acaba de comentar con los sistemas monolíticos, tradicionalmente el kernel no se dividía en varias capas, toda la gestión de recursos y servicios del sistema se hacia en una sola capa en modo privilegiado, por lo que a medida que el kernel iba creciendo era difícil de gestionar y se hacía difícilmente mantenible.

Varios investigadores han estudiado el número de errores por cada 1000 líneas de código. La densidad de los errores depende del tamaño del módulo, pero una cifra aproximada para los sistemas formales es de diez errores por cada mil líneas de código. Esto significa que es probable que un sistema operativo monolítico de cinco millones de líneas de código contenga cerca de 50.000 errores en el kernel.

La idea básica detrás del diseño de microkernel es lograr una alta confiabilidad al dividir el sistema operativo en **varios módulos pequeños y bien definidos**, donde sólo un módulo llamado **microkernel** se ejecuta en modo kernel, y el resto se ejecuta como procesos de usuario ordinarios.

En los sistemas con microkernel, el núcleo del sistema operativo se mantiene en el **mínimo posible de funcionalidad**, descargando en **procesos especiales de sistema sin privilegios** las tareas que implementan el acceso a dispositivos físicos y las diversas políticas de uso del sistema. Es decir, se eliminan todos los componentes no esenciales del kernel y se implementan como programas de sistema y usuario, resultando en un kernel muy pequeño. Es todo lo contrario a lo sistemas monolíticos, donde el kernel lo gestionaba prácticamente todo.

El microkernel proporciona **servicios mínimos**, aunque no hay una definición clara de las funciones que debe llevar a cabo un micronúcleo.:

- Cierta administración de la memoria.

- Una cantidad limitada de planificación y administración de procesos de bajo nivel.
- Entrada/salida de bajo nivel (gestión de interrupciones).

En la siguiente figura se muestra un sistema genérico con microkernel:

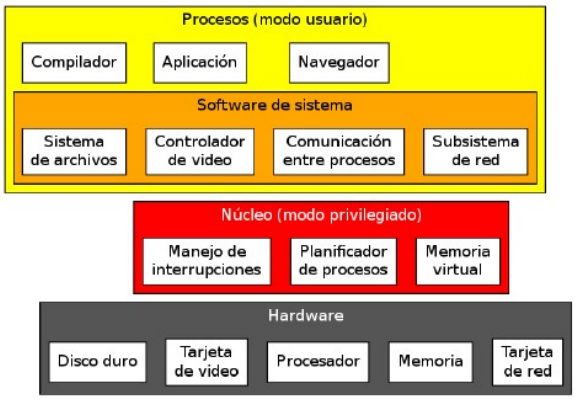


Figura 1.10: Esquematización de los componentes en un sistema microkernel.

A continuación se muestra como ejemplo concreto la estructura del sistema operativo Minix 3 (http://www.minix3.org/), que se organiza en varios niveles de usuario, donde los programas de usuario hacen llamadas a un conjunto de programas "servidores" (realizan la mayor parte del trabajo del sistema operativo) y estos a su vez a otros programas llamados "drivers", que son los que se comunican con el núcleo o microkernel. Esta división también se puede encontrar en la literatura nombrada como **sistemas cliente-servidor**, en el sentido de que un proceso de usuario, denominado proceso cliente, solicita un servicio a un proceso servidor. A su vez, el proceso servidor puede requerir los servicios de otros servidores.

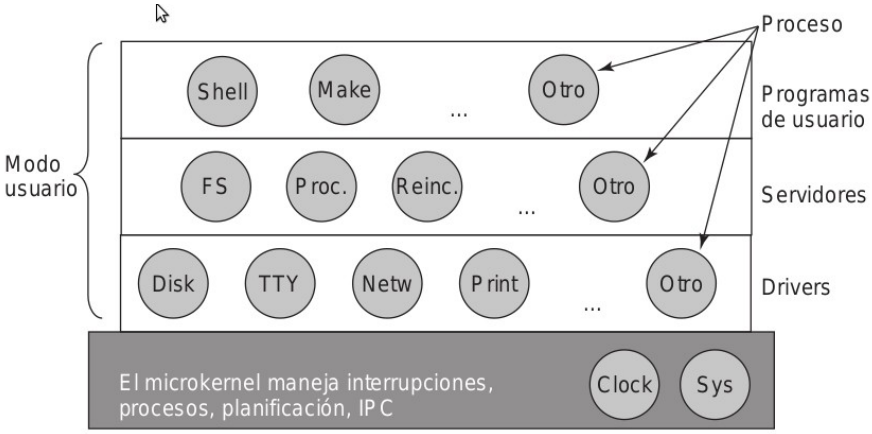


Figura 1-26. Estructura del sistema MINIX 3.

El sistema de microkernels tiene algunas **ventajas**, como la facilidad para **ampliar el sistema** operativo. Todos los servicios nuevos se añaden al espacio de usuario y, en consecuencia, no requieren que se modifique el kernel. Cuando surge la necesidad de modificar el kernel, los cambios tienden a ser pocos, porque el microkernel es un kernel muy pequeño. El sistema operativo resultante es más fácil de portar de un diseño hardware a otro.

El microkernel también proporciona **más fiabilidad** en cuanto a que la mayor parte de los servicios se ejecutan como procesos de usuario, en lugar de como procesos del kernel. Si un servicio falla, el resto del sistema operativo no se ve afectado.

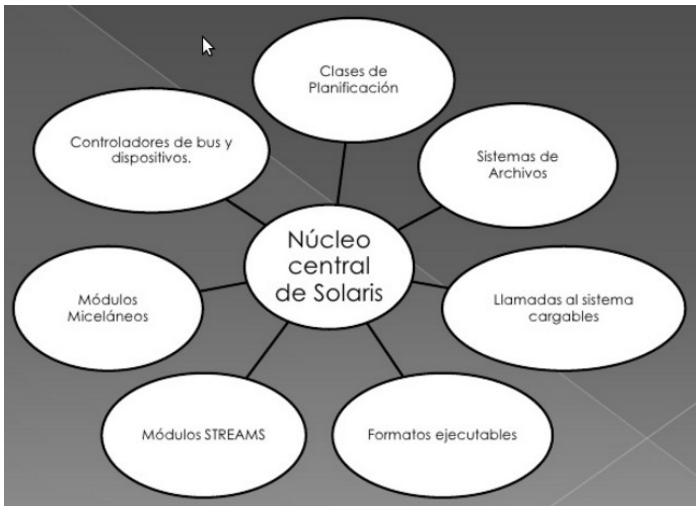
Lamentablemente, los microkernels pueden tener **un rendimiento peor** que otras soluciones, debido a la **carga de procesamiento adiciona**l impuesta por las funciones del sistema. Esto se debe a que los distintos componentes de un sistema operativo de este tipo ejecutan en espacios de direcciones distintos, lo que hace que la activación de cada proceso requiera más tiempo.

Consideremos la historia de Windows NT: la primera versión tenía una organización de microkernel con niveles, sin embargo, esta versión proporcionaba un rendimiento muy bajo, comparado con el de Windows 95. La versión Windows NT 4.0 solucionó parcialmente el problema del rendimiento, pasando diversos niveles del espacio de usuario al espacio del kernel e integrándolos más estrechamente.

11.3 Sistemas basados en Módulos

Quizá la mejor metodología actual para diseñar sistemas operativos es la que usa la idea de **programación orientada a objetos** 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 resulta habitual en las implementaciones modernas de UNIX, como Solaris, GNU/Linux y Mac OS X.

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 de dispositivos para hardware específico y puede agregarse como módulo cargable el soporte para diferentes sistemas de archivos.



El resultado global es similar a un sistema de varios niveles, en el sentido de que cada sección del kernel tiene interfaces bien definidas y protegidas, pero es **más flexible que un sistema de niveles**, **porque cualquier módulo puede llamar a cualquier otro módulo**. Además, el método es similar a la utilización de un microkernel, ya que el **módulo principal sólo dispone de las funciones esenciales** y de los conocimientos sobre cómo cargar y comunicarse con otros módulos.

Un módulo cargable del núcleo es un archivo que contiene código objeto que puede extender el núcleo en ejecución. Los módulos cargables en el núcleo son generalmente utilizados para **brindar soporte a nuevos dispositivos de hardware** (actúan de drivers software) y sistema de archivos, así como para agregar llamadas al sistema. Cuando la funcionalidad provista por un módulo del núcleo deja de ser requerida, normalmente éste puede ser descargado, **liberando su memoria**.

Un sistema operativo que **no dispone de módulos cargables** en el núcleo debe tener toda aquella funcionalidad que pueda llegar a ser necesitada precompilada dentro del núcleo base. El problema de este enfoque consiste en que en general la imagen del núcleo sería mucho mayor, ocupando un gran espacio memoria. Así mismo, resultaría necesario que los usuarios **recompilaran** y reiniciaran el núcleo base cada vez que se necesite agregar nueva funcionalidad al mismo.

Si bien los módulos cargables consisten en un método conveniente para modificar el núcleo en ejecución, esto podría llegar a ser abusado por un atacante en un sistema comprometido (**problemas de seguridad**), con el fin de prevenir la detección de procesos o archivos maliciosos, permitiéndole mantener control sobre el sistema o robar información privada. En el caso de GNU/Linux, la mayoría de los controles de seguridad del sistema se refuerzan a través del núcleo. Si éste tiene problemas de seguridad, el sistema completo será vulnerable. En la distribución oficial del núcleo, solo un usuario autorizado, conocido como el superusuario (***root***), puede cargar módulos.