El SO configura periódicamente una interrupción de reloj vía hardware para evitar perder el control y cambiar de modo, de forma que un usuario no pueda acaparar todos los recursos del sistema. También el planificador del sistema se encarga de "repartir" los recursos del mismo en función de la política que se use.

10 Llamadas nativas al sistema

Una llamada nativa al sistema es utilizada por una aplicación (programa de usuario) para **solicitarle un servicio al sistema operativo,** por ejemplo, la apertura o cierre de un fichero, la escritura de un mensaje por la salida estandar del sistema. Normalmente se ejecutan **miles de llamadas nativas al sistema por segundo**, ya sea a partir de programas de usuario o por el propio sistema operativo para realizar tareas de gestión.

Desde este punto de vista, las llamadas nativas al sistema **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.

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

1. **INDIRECTAMENTE:** 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 al sistema disponibles.

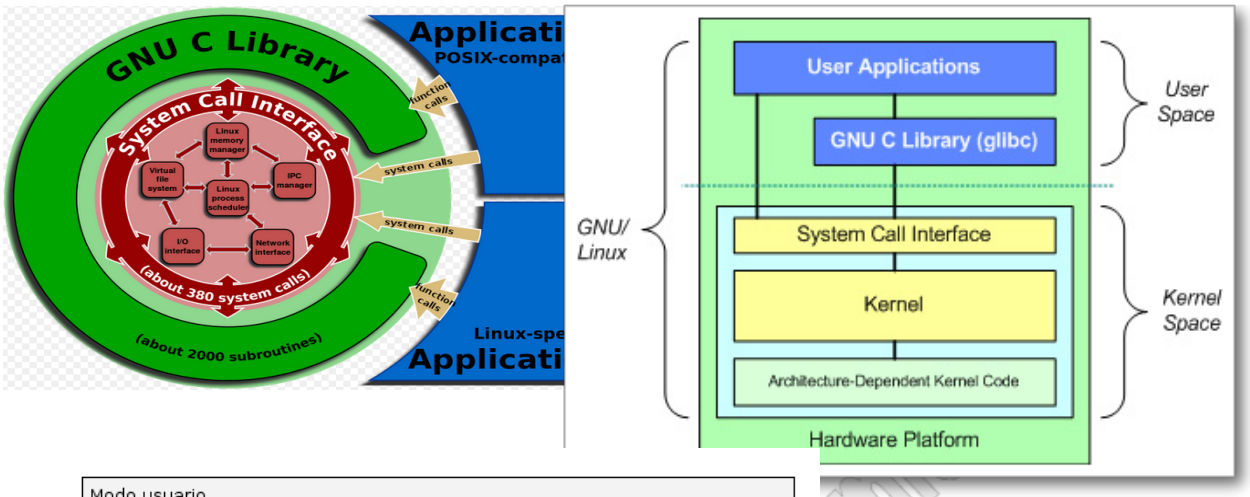
   Dos de las API más usuales disponibles para programadores de aplicaciones son la API Win32 para sistemas Windows y la API de la biblioteca estándar de C, *glibc,* para sistemas basados en POSIX (prácticamente todas las versiones de Gnu/Linux y Mac OS).

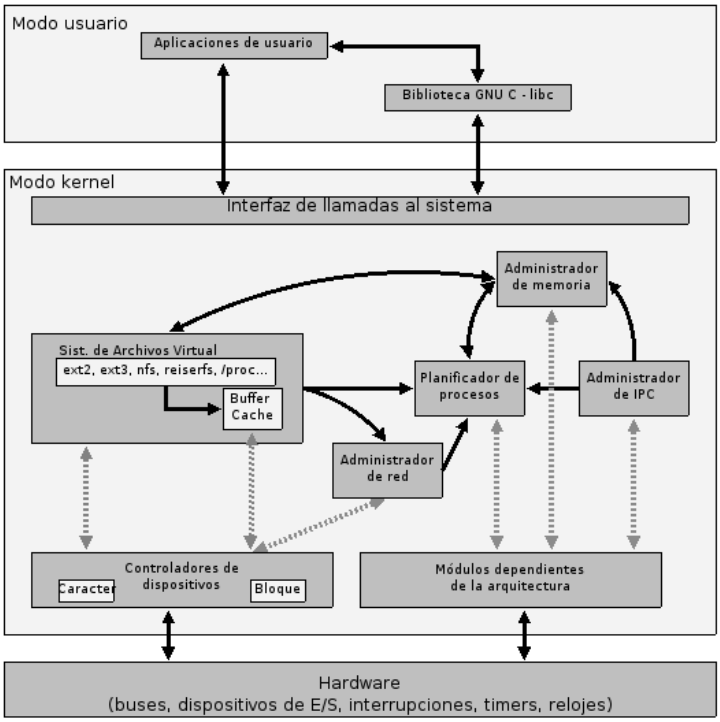
   En la bibliografía puede encontrar que se nombre llamada al sistema a una función de una API, como por ejemplo ***printf()*** o ***write()*,** pero se hace para simplificar, ya que realmente eso no son llamadas al sistema. Las funciones que conforman una API de alto nivel invocan a las llamadas nativas al sistema internamente a través de una función nombrada TRAP (se estudiará después), por lo que realmente actúan como ***wrapers*** o envolventes de **funciones nativas del núcleo**. Es decir, la propia función ***printf()*** invocada no es la llamada nativa al sistema en si, sino que por debajo se encuentra una llamada o un conjunto de llamadas nativas al sistema. 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 la función *write()* de Gnu/Linux, que invoca a una función nativa llamada *__NR_write*.

2. **DIRECTAMENTE:** Acceso **mediante la interfaz de llamadas nativas ofrecida por el núcleo del sistema operativo**, mediante lenguaje **ensamblador**. Esto es una manera más compleja de usar las llamadas nativas 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.

Las siguientes figuras muestran gráficamente dónde se sitúan las llamadas nativas al sistema y las "llamadas al sistema" a través de API:





- Biblioteca GNU C – lib (modo usuario): *printf()*, *write()*, *getpid()*, *open()*, *close()*, *fopen()*, *fclose()*, etc
- Interfaz de llamadas nativas al sistema (modo núcleo): __NR_write, __NR_fork, __NR_exit, __NR_open, __NR_close, etc

Cuando se realiza una llamada nativa al sistema puede ser necesario **pasar parámetros** al núcleo del sistema operativo desde la llamada en la API. 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 que son accesibles en modo usuario** y que posteriormente serán copiados al núcleo.
2. En algunos casos, puede haber más parámetros que registros disponibles. En estos casos l**os parámetros se almacenan en un bloque en memoria a nivel de usuario**, y la **dirección del bloque** se pasa como parámetro a un registro del procesador que posteriormente será copiado al núcleo.
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 y **copiarlos a otra pila a nivel de núcleo**. 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 nativa al sistema tiene asociado un número** y la interfaz de llamadas nativas al sistema mantiene una **tabla indexada en la memoria** con dichos números. Usando esa tabla se invoca la llamada nativa necesaria del ***kernel*** del sistema operativo y devuelve el **estado de la ejecución de dicha llamada nativa + 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 nativa a sistema **no se identifica con una dirección, sino con un identificador** (un número que se debe conservar constante entre versiones). Por tanto, la tabla de llamadas nativas al sistema es la que ofrece compatibilidad entre diferentes versiones del SO.

10.1 Funciones de tipo trap para acceso al núcleo

Una vez presentado el concepto de llamada nativa 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 APIs:

1. Cuando se produce una llamada nativa 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 accesibles** a nivel de usuario, dependiendo del sistema**.**
2. Posteriormente, en la implementación interna de la función API invocada, se ejecuta una función especial 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 en cuanto a abstracción, ya que estas funciones tipo *trap* no forman parte de una API como tal, sino que son las funciones de la propia API las que la invocan internamente (todavía en modo usuario).

3. La función de tipo ***trap***, entre otras cosas, se encargará de obtener la dirección de los parámetros almacenados en la llamada a nivel de API, y de ejecutar una **interrupción especial** que conlleva **un cambio de modo**, de modo usuario a modo núcleo o supervisor (**NO** tiene nada que ver con el superusario ***root***). Este cambio de modo se realiza por hardware cuando se invoca a la interrupción especial.
4. Siempre que se realiza una llamada al sistema hay que **guardar el estado** de ejecución actual y el valor del PC, ya que se va a saltar a una zona de memoria donde se encontrará alojada la llamada nativa del sistema a invocar y después no se sabría volver al proceso que se estaba ejecutando. Para ello, la interrupción especial invocada por la función de tipo ***trap*** hará que se carguen ya en modo núcleo otras rutinas para hacer estas operaciones de salvado, justo **antes de empezar a ejecutar la llamada nativa al sistema** concreta a partir de su número de identificación.
5. Posteriormente al salvado se **examina el identificador de la llamada nativa invocada y sus parámetros**, y se busca en una tabla o vector de rutinas si existe dicho 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 nativa 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 nativa al sistema se invocan otras subrutinas denominadas de tipo **RETURN FROM TRAP**, para:
   - **Devolver el código de estado** de la llamada nativa al sistema (OK, ERROR).
   - **Devolver los posibles parámetros de retorno** que pudiera devolver la misma.
   - **Restaurar el contexto** del proceso salvado, el que realizó la llamada al sistema a través de API.
   - Pasar **de modo núcleo a a modo usuario**.
8. Cuando finalizan esas acciones se regresa el control a la función de API (*printf()*, *write()*, etc) y **a nivel de usuario se descarga la pila** y se comprueba el resultado de la ejecución de la petición al sistema.

Tenga en cuenta que el esquema proporcionado es genérico y que no se ha tenido en cuenta que la ejecución de la llamada nativa al sistema se pueda interrumpir. Dependiendo del sistema y su configuración o políticas del núcleo, una excepción, una interrupción o una llamada nativa 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 nativa 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 nativa 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 nativa 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 nativas al sistema en GNU/Linux

A continuación se muestra como se lleva a cabo el mecanismo mediante el cual GNU/Linux implementa las **llamadas nativas al sistema en una arquitectura clásica x86 (32 bits)**. Tenga en cuenta que dependiendo de la arquitectura y dependiendo de la versión del sistema operativo y su kernel, la interfaz de llamadas nativas al sistema puede variar, así como el procedimiento llevado a cabo para realizarlas..

Cada sistema operativo usa el hardware de la computadora de una manera específica. Por tanto, para un mismo hardware dos sistemas operativos distintos mostrarán un interfaz de llamadas nativas al sistema distinto. Para saber cómo se producen internamente las llamadas nativas 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 (recuerde, no son nativas todavía) 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 nativa al sistema.**

Para la arquitectura y versión de núcleo comentada, en el fichero ***include/asm-i386/unistd.h*** (cuidado, hay varios ficheros nombrados así GNU/Linux) aparecen listadas las llamadas nativas al sistema que se ofrecen con su correspondiente número, por ejemplo **__NR_close**, que corresponde a la llamada nativa al sistema ***sys_close()*** (lo usaremos indistintamente), que es la función ***close()*** a nivel de API.

```
/*
 * This file contains the system call numbers.
 */

#define __NR_restart_syscall            (__NR_SYSCALL_BASE+  0)
#define __NR_exit                       (__NR_SYSCALL_BASE+  1)
#define __NR_fork                       (__NR_SYSCALL_BASE+  2)
#define __NR_read                       (__NR_SYSCALL_BASE+  3)
#define __NR_write                      (__NR_SYSCALL_BASE+  4)
#define __NR_open                       (__NR_SYSCALL_BASE+  5)
#define __NR_close                      (__NR_SYSCALL_BASE+  6)
                                        /* 7 was sys_waitpid */
#define __NR_creat                      (__NR_SYSCALL_BASE+  8)
#define __NR_link                       (__NR_SYSCALL_BASE+  9)
#define __NR_unlink                     (__NR_SYSCALL_BASE+ 10)
#define __NR_execve                     (__NR_SYSCALL_BASE+ 11)
#define __NR_chdir                      (__NR_SYSCALL_BASE+ 12)
#define __NR_time                       (__NR_SYSCALL_BASE+ 13)
#define __NR_mknod                      (__NR_SYSCALL_BASE+ 14)
#define __NR_chmod                      (__NR_SYSCALL_BASE+ 15)
#define __NR_lchown                     (__NR_SYSCALL_BASE+ 16)
                                        /* 17 was sys_break */
                                        /* 18 was sys_stat */
 ... mas llamadas al sistema ...

#define __NR_pipe2                      (__NR_SYSCALL_BASE+359)
#define __NR_inotify_init1              (__NR_SYSCALL_BASE+360)
#define __NR_preadv                     (__NR_SYSCALL_BASE+361)
#define __NR_pwritev                    (__NR_SYSCALL_BASE+362)
#define __NR_rt_tgsigqueueinfo          (__NR_SYSCALL_BASE+363)
#define __NR_perf_event_open            (__NR_SYSCALL_BASE+364)
```

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

- Esta interrupción **se invoca a partir de la rutina *__init trap_init()***, que 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 de la función ***__init trap_init()*** se establece el método de entrada al núcleo mediante la función ***set_system_gate(*SYSCALL_VECTOR,&*system_call),*** que invoca a la interrupción ***int 0x80*** y se 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)***.

Recuerde que de manera general y antes de entrar en modo núcleo se habrán hecho otras operaciones como por ejemplo recoger

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 }
```

En esa llamada ***set_system_gate(*SYSCALL_VECTOR,&*system_call)*** se produce un cambio de modo, **ahora en modo núcleo los pasos son los que se exponen a continuación.** Para intentar dar una mejor comprensión del complejo proceso de llamadas nativas 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 **identificador de la llamada nativa al sistema** que se quiere invocar, recogido del registro ***%eax*** del procesador, que se habrá cargado en algún paso anterior (todavía en modo usuario) a la invocación de ***set_system_gate(*SYSCALL_VECTOR,&*system_call)***, además de otra información como los **parámetros de entrada, dirección de retorno**, etc (se hace una **copia** en el núcleo del último registro de activación de la **pila a nivel de usuario**). En el siguiente tema se recordará en más profundidad el uso de la pilas y la información que se almacena en ellas.

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.***

3. Se comprueba que el número de llamada nativa 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 en ***errno***.

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

   La llamada nativa al sistema tiene esta forma de prototipo: ***sys_nombre-de-la-funcion(parámetros).*** Los parámetros que se recogen de la pila de la función ***system_call(),*** ya se copiaron anteriormente al espacio del núcleo**.**

5. Al finalizar la ejecución de la llamada nativa 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. En caso contrario se pasará el proceso interrumpido a la lista de listos y se cambiará al contexto de otro proceso que indique el planificador.

Para restaurar el proceso de usuario que hizo la llamada a nivel de API 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.**

6. Para volver a modo usuario se salta al código de la **subrutina *resume_userspace*,** también situado en el archivo en ensamblador ***arch/x86/kernel/entry_32.S*** y se ejecuta una instrucción de retorno de interrupción de tipo ***iret***.

   Se restauran los valores necesarios en la pila a nivel de usuario (incluido el valor devuelto por la función, que está almacenado en ***%eax***), y se cambia a modo usuario.

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

8. Cuando la llamada no ha tenido éxito, el valor devuelto a través de ***%eax*** es negativo (***-1***) y es necesario consultar la variable global ***errno*** (contiene el código de error de la última llamada que falló). Una llamada API que se realice con éxito no modifica ***errno.***

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