CENTRO UNIVERSITARIO
DE TECNOLOGÍA
Y ARTE DIGITAL

Sistemas Operativos

6. Entrada/Salida

- Javier García Algarra
- javier.algarra@u-tad.com

- 2021-2022

Introducción

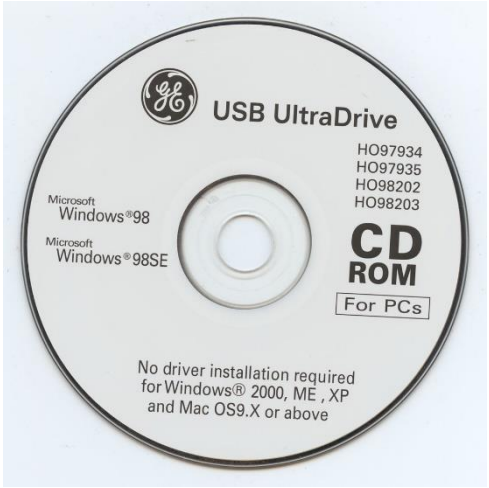
Toda la relación del ordenador con su entorno se realiza a través de los dispositivos de Entrada/Salida, que pueden ser tan variados como un ratón USB, un juego de altavoces o el servomotor que controla el alerón de un avión.

En el capítulo de introducción, ya se indicó que todas las operaciones que involucran a estos dispositivos tienen que pasar forzosamente por el Sistema Operativo ya que el código necesario se ejecuta en modo privilegiado.

Dennis Ritchie (de pie) y Ken Thompson trabajando desde la consola de un equipo PDP-11, un ejemplo de dispositivo de E/S. No, no tenía monitor, la salida era en papel, era un teletipo.

Introducción

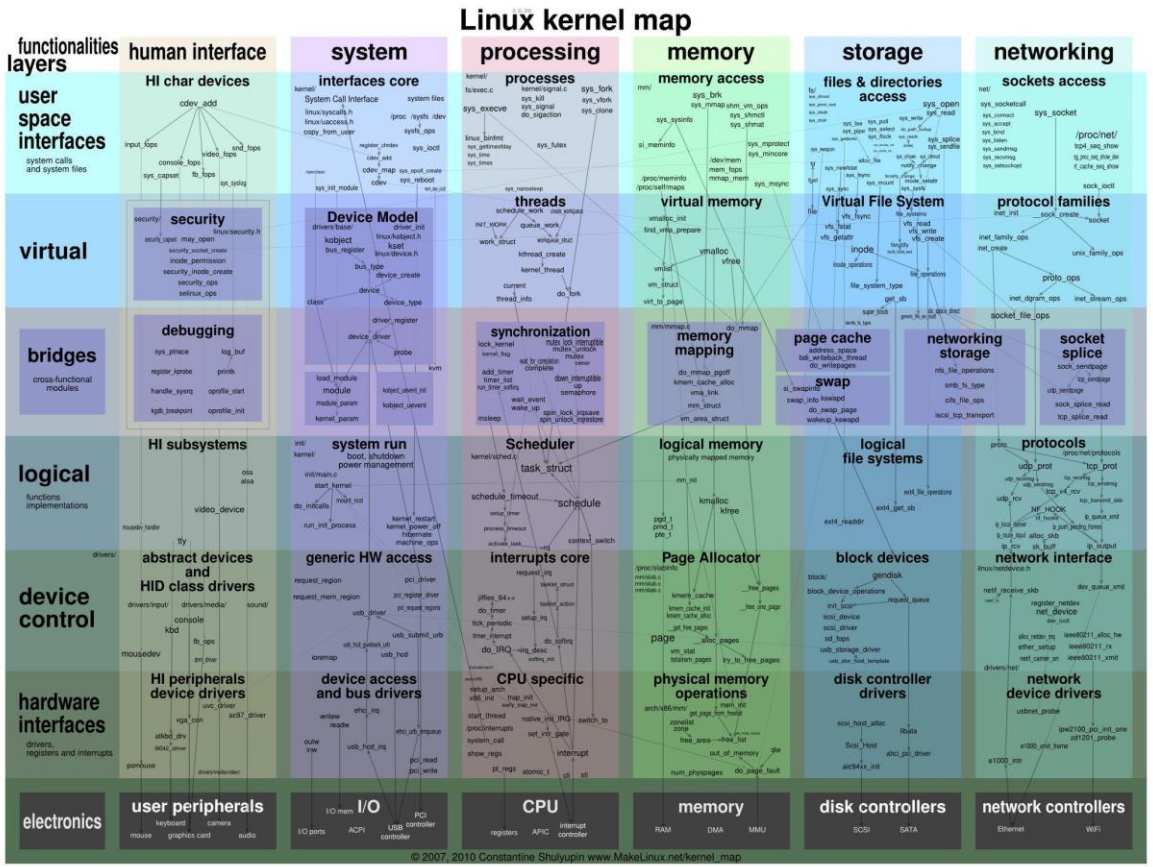
Como en las anteriores lecciones, veremos que el Sistema Operativo oculta los detalles más complejos del manejo de estos equipos electrónicos mediante la abstracción **dispositivo** (*device*), que es una ampliación de la abstracción **fichero**.



La capa más próxima a los dispositivos son los **manejadores** (*drivers*). Dada la gran variedad de equipos en el mercado, no es de extrañar que un porcentaje muy elevado de los núcleos de Windows (50 millones de líneas de código) o Linux (15 millones), corresponda a los drivers. Este software suelen desarrollarlo los mismos fabricantes de los equipos, debido a que son quienes mejor conocen las complejidades de sus productos.

Introducción

Busca los drivers en la imagen



Introducción

Típico ingeniero júnior en su primera crisis vital.

Esto es horrible. Si el kernel solo puede ejecutarse en modo supervisor y mis programas, forzosamente, en modo usuario, nunca podré desarrollar un *driver* por mi cuenta. Solo seres semidivinos como los que mantienen el kernel o los que escriben los drivers de las GPUs de NVIDIA pueden hacer ese tipo de cosas. Me pasaré los próximos cuarenta años instalando parches en aplicaciones COBOL escritas en tiempos de mi abuelo.

.

¡¡¡ He malgastado mi vida!!! Debería haber atendido mejor en clase de Sistemas Operativos.

Introducción

Alumna que escribió su primer “hello driver” en clase de Sistemas Operativos y triunfa en la vida.

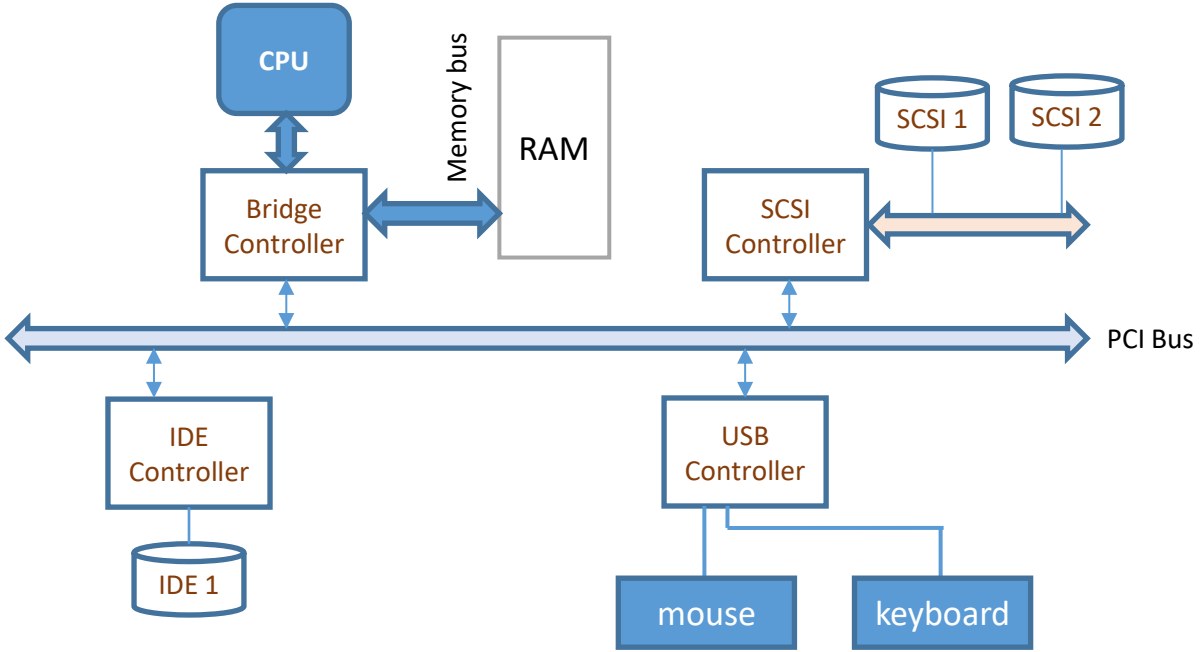
En efecto, si atendieseis más en clase, vuestras vidas profesionales serían más felices. Aun estáis a tiempo.

Es cierto que el kernel se ejecuta en modo supervisor, pero tiene la capacidad de cargar módulos dinámicos que se ejecutan en modo supervisor. Esto es válido para Windows, Linux y FreeBSD, por citar tres ejemplos. No hace falta ser un super héroe ni pedirle la password de su equipo a Linus Torvalds o Bill Gates para poder generar código en modo privilegiado.

En la parte práctica de esta lección aprenderás cómo hacerlo.

Modelo de entrada/salida

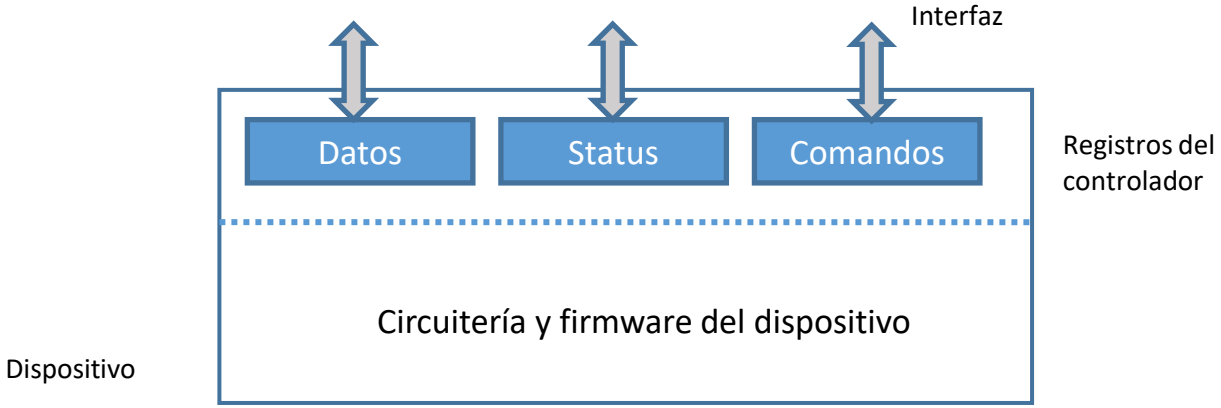
Todos los dispositivos se comunican con la CPU usando protocolos que consisten en el envío de mensajes y señales eléctricas. Si la conexión es compartida por varios dispositivos se denomina **bus**.



Modelo de entrada/salida

La CPU se comunica con distintos tipos de bus, usando circuitos de arbitraje que se llaman bridges (¿recuerdas North Bridge y South Bridge?). Con independencia de sus detalles físicos y funcionales todos los dispositivos se comunican con el mundo exterior usando un circuito llamado controlador.

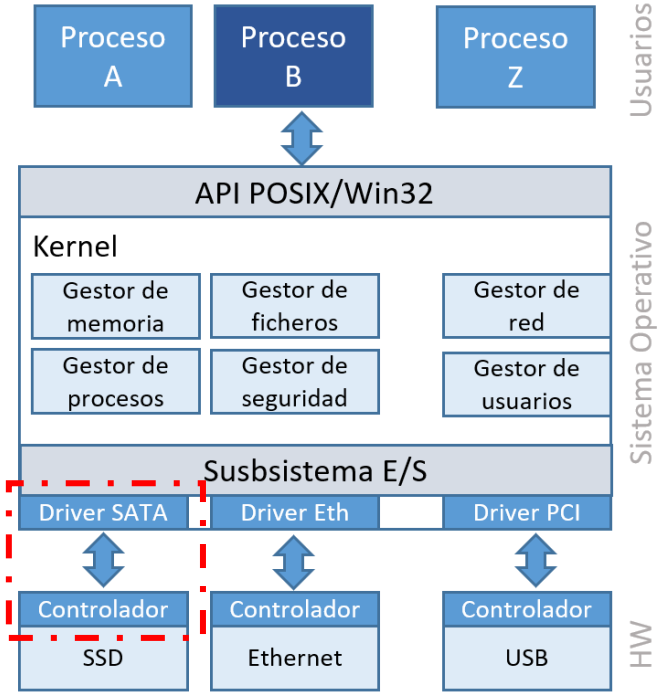
Un controlador es en sí mismo un microordenador empotrado en el dispositivo. Un disco SSD, una GPU o un modesto ratón USB incorporan una capacidad de cómputo notable para llevar a cabo su cometido, pero su vida interior es opaca para el sistema operativo (no digamos para el usuario). La relación con los dispositivos se realiza usando un pequeño conjunto de registros.



Modelo de entrada/salida

¿Cómo sería la escritura en un fichero ya abierto? Vamos a suponer un modelo de disco que no existe por ineficiente, pero que permite escribir carácter a carácter.

El driver recibe los datos de las capas superiores, ahora no nos interesan los detalles, y escribe el siguiente carácter en el registro de datos del controlador. Este registro está fuera de la CPU, el mapeo de la dirección depende del modelo de microprocesador, nos basta con saber que el driver puede leer y escribir en ese registro. A continuación pondrá en el registro de comandos la orden de escritura, su forma concreta dependerá del equipo, puede ser un código hexadecimal. Pasado un tiempo aparecerá en el registro de status el resultado de la operación, sea correcto o erróneo.



Modelo de entrada/salida

El controlador simplifica mucho la vida al programador del Sistema Operativo puesto que oculta toda la complejidad interna del dispositivo. El programador de aplicaciones tiene aun una visión mucho más sencilla porque cuando accede a un fichero le resulta indiferente que el soporte sea magnético, SSD u óptico y si se conecta mediante interfaz IDE o SCSI.

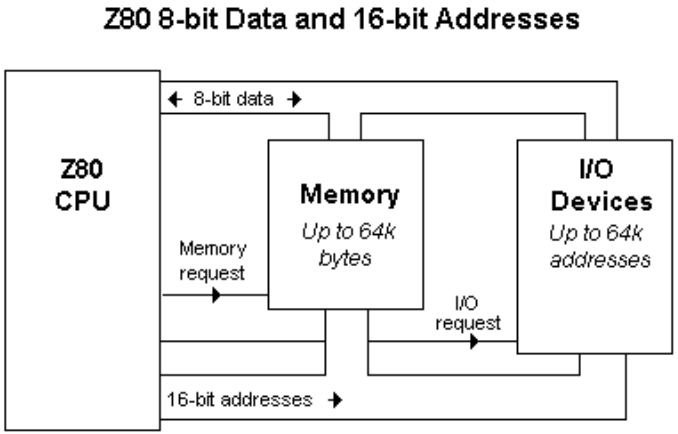
La comunicación desde el SW atraviesa varias capas del kernel hasta llegar al driver adecuado, la relación entre dispositivos físicos y lógicos la mantiene este. Por ejemplo, sabe que el descriptor de fichero número 124 corresponde a una partición del disco IDE 1.

En el ejemplo de la escritura de un carácter usando los registros del controlador aparecen dos de los problemas que tiene que resolver el Sistema Operativo en la relación con los dispositivos de entrada/salida. El primero es el direccionamiento, ¿cómo sabe la CPU cuál es el registro de datos o de status de un dispositivo determinado y cómo accede a él? El segundo, es la adaptación de velocidades. En lo que el disco escribe un carácter la CPU ha podido ejecutar millones de instrucciones. ¿Qué hace el driver, se queda en espera?

Modelo de entrada/salida

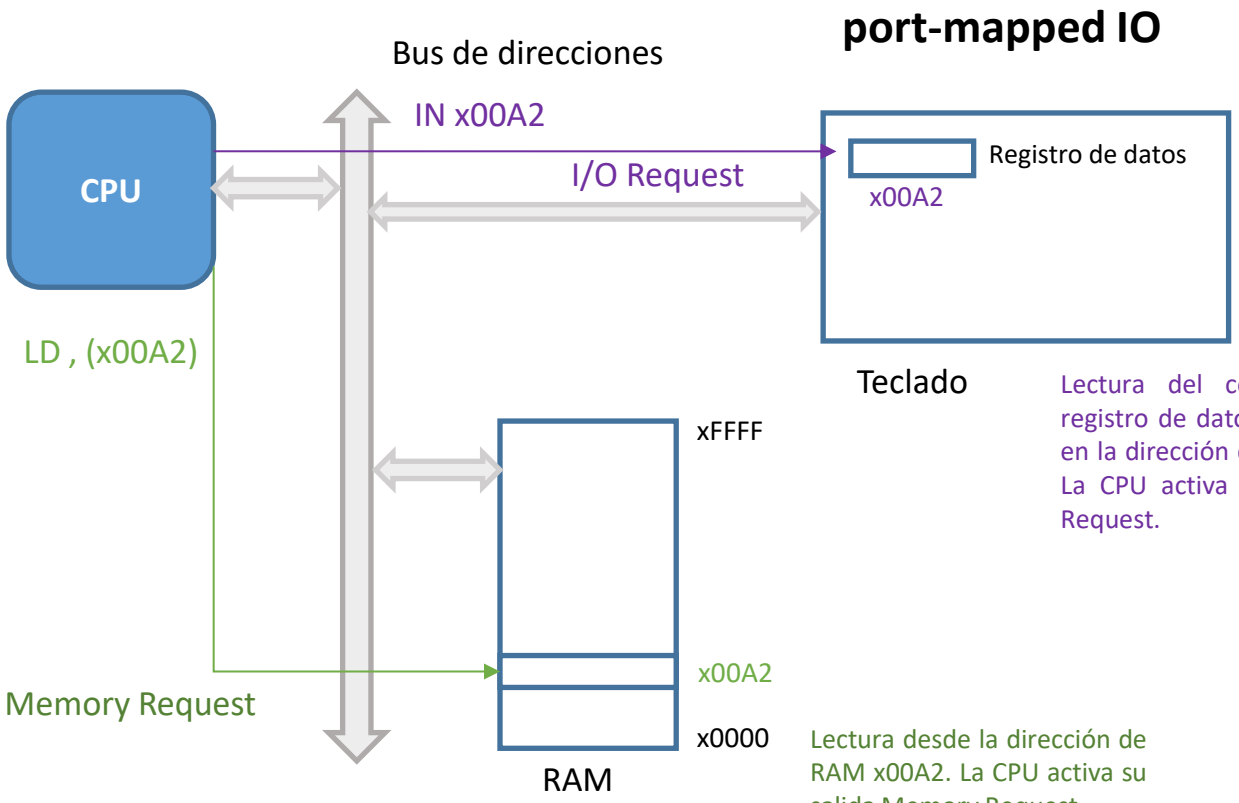
Direccionamiento

A lo largo del tiempo se han usado dos tipos de direccionamiento. Algunos procesadores usan direccionamiento e instrucciones específicas para los dispositivos de entrada/salida, como en el modelo de máquina de von Neumann. Esta solución tenia sentido para las máquinas antiguas en las que el mapa de memoria era muy reducido y las direcciones un recurso escaso. Es lo que se conoce como puertos de memoria mapeados (**port-mapped IO**). Se sigue usando en microcontroladores.



El Z80 es un ejemplo de micro con *port-mapped IO.*
Cuando el micro tiene que comunicarse con un dispositivo escribe la dirección en el bus y activa la señal I/O request.

Modelo de entrada/salida



Lectura del contenido del registro de datos del teclado en la dirección de E/S x00A2. La CPU activa su salida I/O Request.

Lectura desde la dirección de RAM x00A2. La CPU activa su salida Memory Request.

Modelo de entrada/salida

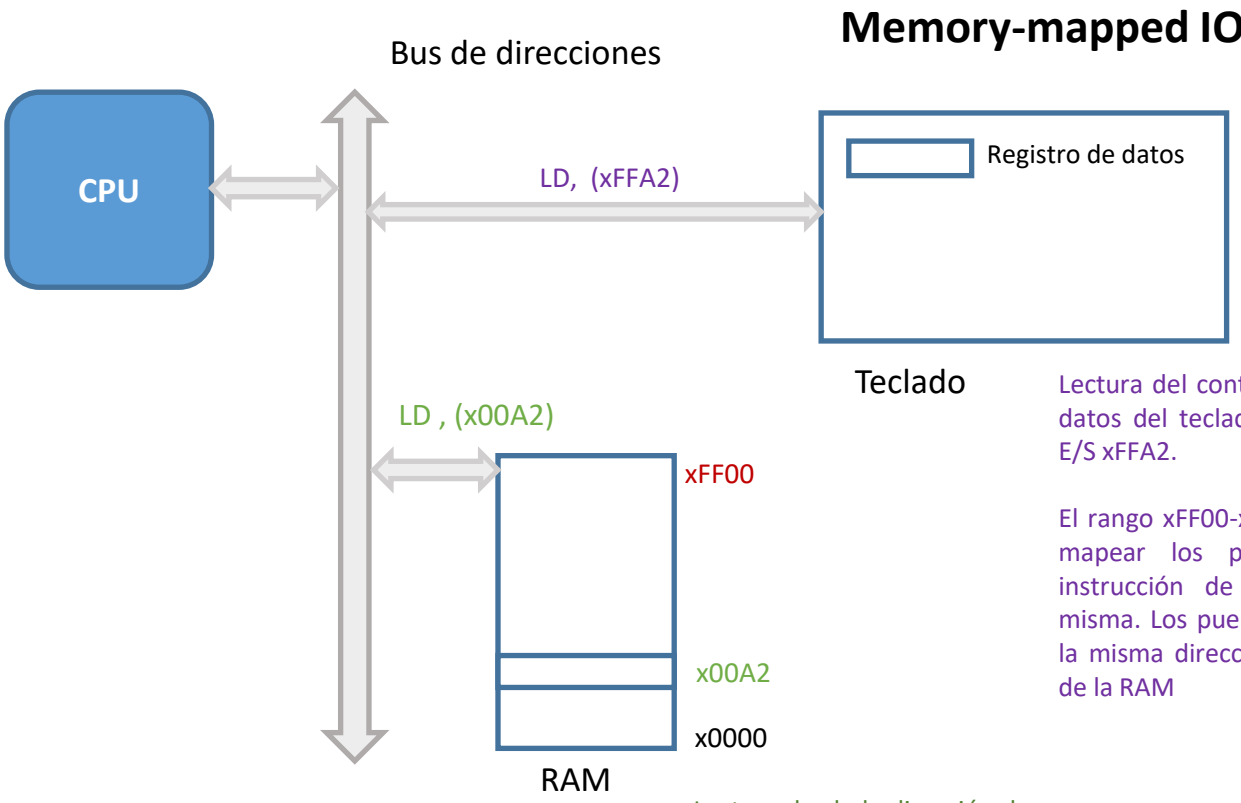
Direccionamiento

Con 32 bits de direcciones ya no había ningún problema en reservar un pequeño rango fijo de direcciones de memoria en las que mapear los registros de entrada/salida de los dispositivos (**memory-mapped IO**). Con este montaje, el programador no tiene que usar instrucciones de código máquina distintas si accede a la RAM o al registro de un controlador.

La arquitectura x86 mantiene el modo de puertos mapeados original del 8086, pero también permite este funcionamiento.

Hay dos barreras que superar para que este modo funcione correctamente. La primera es la caché. Los dispositivos están mapeados en direcciones físicas, pero la caché se interpone. Para evitarlo, en la tabla de páginas del proceso esas páginas aparecen con el bit *caching disabled* activado que fuerza la inactivación de ésta por el Sistema Operativo.

Modelo de entrada/salida



Lectura del contenido del registro de datos del teclado en la dirección de E/S xFFA2.

El rango xFF00-xFFFF se reserva para mapear los puertos de E/S. La instrucción de ensamblador es la misma. Los puertos no pueden tener la misma dirección que una posición de la RAM

Lectura desde la dirección de RAM x00A2.

Modelo de entrada/salida

Modos de programación E/S

La adaptación de velocidades entre la CPU y el dispositivo es un problema serio. La solución más sencilla e ineficaz es la espera activa.

Volvamos al ejemplo del disco en el que se escribe carácter a carácter. El driver escribe el carácter en el registro de datos y la orden de escritura en el registro de comandos. Pasarán muchos ciclos de reloj hasta que el registro de status devuelva el resultado de la operación.

La espera activa consiste en que el driver lee en un bucle infinito el registro de status hasta que cambia su contenido. Poco elegante, ¿no es cierto?

Modelo de entrada/salida

Ejemplo de espera activa vs espera bloqueante. Este es el seudocódigo de un manejador que va escribiendo caracteres en un dispositivo con tres registros de datos, operación y status.

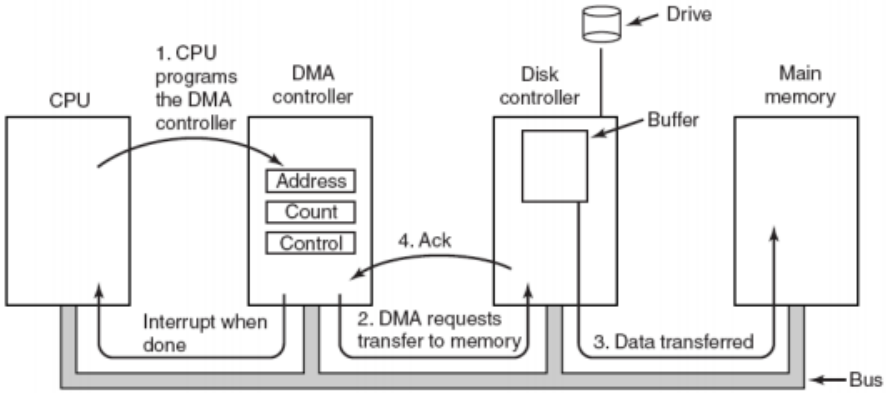
```
i = 0;
do {  write_reg(reg_data,texto[i]);
      write_reg(reg_opcode, WRITE);   //Orden de escritura
      while (reg_status == BUSY) ;    //Bucle de espera, quemando CPU
      i++;}
while (texto[i]!='\0');              //Escribir todo el string

i = 0;
do {  write_reg(reg_data,texto[i]);
      write_reg(reg_opcode, WRITE);   //Orden de escritura
      sleep_wait()                    // A bloqueo
                                      // Desbloqueado por el SO
                                      // Transparente al programador
      i++;}
} while (texto[i]!='\0');
```

Modelo de entrada/salida

Modos de programación E/S

La entrada salida bloqueante tiene como inconveniente los cambios de contexto. Si el dispositivo genera poco tráfico de datos como un ratón o un teclado no es problema, pero si se trata de transferir grandes bloques de información entre el disco y la RAM o entre la GPU y el monitor, la estrategia no funciona. Para este escenario se inventó el **DMA** (Direct Memory Access). La idea básica del DMA es evitar que los datos de estas transferencias masivas pasen por la CPU, sino que fluyan entre los dos dispositivos de una forma transparente. Esto es solo posible con hardware especial, el controlador de DMA.



Modelo de entrada/salida

Modos de programación E/S

Además del controlador de DMA los dispositivos involucrados también tienen que soportar este método. Supongamos que un proceso quiere cargar un fichero grande (una imagen) que ocupa miles de bloques. La CPU programa el controlador de DMA indicando en qué dirección quiere leer el primer bloque e indica al controlador de disco que lea ese bloque en su memoria interna. A partir de ese momento la CPU se inhibe, el controlador de DMA envía una orden al controlador de disco y comienza la transferencia entre los dos sin pasar por la CPU. Obviamente, ocuparán el bus de datos. Para no colisionar con las tareas que esté realizando la CPU se emplean dos técnicas:

- Transferencia a ráfagas. Desde que comienza la transferencia hasta que termina el bus queda en poder de los dispositivos. La CPU puede seguir trabajando si es un periodo breve con los datos de la caché.
- Robo de ciclo. Cada N ciclos de bus, el controlador del dispositivo origen (el disco en este caso) activa una señal de “robo”, un protocolo por el que se solicita permiso de acceso a la CPU y esta se desconecta del bus mientras dura la operación. Reparte mejor los posibles tiempos de espera de la CPU.

Modelo de entrada/salida

Modos de programación E/S

Cuando termina la transferencia, el controlador de DMA interrumpe a la CPU. El *kernel* entra en acción y si la operación requiere la transferencia de un nuevo bloque se repite el ciclo.

Importante: En DMA hay que manejar direcciones físicas, no virtuales, ya que los dispositivos de E/S no entienden de memoria virtual. Se pueden transferir varios bloques en una misma operación pero solo si las direcciones físicas son contiguas.

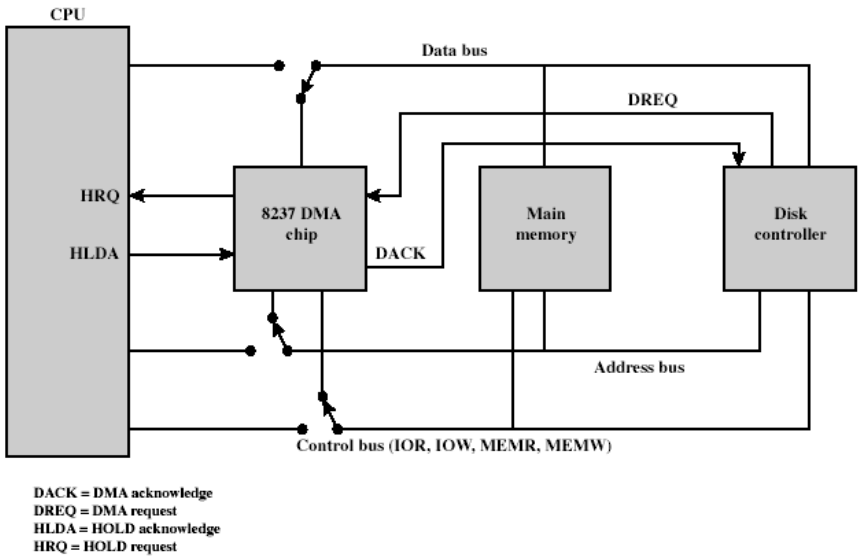


Diagrama de bloques de DMA en Intel, con las señales HRQ y HLDA que arbitran el acceso al bus de la CPU y del controlador de DMA 8237

Dispositivos de Entrada/Salida

La variedad de dispositivos sería inabarcable sin mecanismos de abstracción. En Unix, la entidad *device* puede ser de tres clases:

- **Dispositivos orientados a bloque**: Transfieren la información en bloques de información de tamaño fijo y además permiten acceso aleatorio. El ejemplo característico son las unidades de almacenamiento masivo (discos).

- **Dispositivos orientados a carácter**: En origen, eran aquellos que enviaban o recibían secuencias de caracteres hacia o desde la CPU, como un teclado o una impresora. No son de acceso aleatorio sino que se manejan con colas (buffers) donde se van almacenando los caracteres y solo se puede leer el primero de ellos. Con el tiempo estos dispositivos se han ido volviendo más sofisticados y permiten modos de transferencia más ricos, pero siguen estando en la categoría de orientados a carácter.

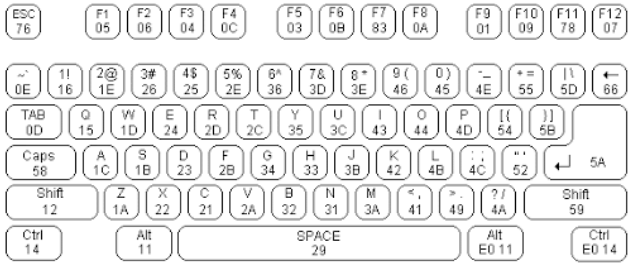
- **Dispositivos orientados a red**: Nacieron más tarde, para atender los dispositivos de red, que tienen características mixtas. Manejan paquetes de información de tamaño fijo pero el acceso no es aleatorio, no podemos saber de antemano cuantos van a llegar en un periodo dado. A esta categoría pertenecen los adaptadores Ethernet o WiFi.

Dispositivos de Entrada/Salida

El terminal

El terminal es el dispositivo de interacción con el usuario por excelencia. Consta de dos equipos lógicos, el teclado y el monitor. En los equipos antiguos, cada vez que el usuario pulsaba una tecla se producía una interrupción y la CPU recibía el código de teclado correspondiente, que después convertía a ASCII o EBCDIC. Para borrar la última letra, se tenía que recibir el código de la tecla borrar y procesarlo en la CPU. La llegada de códigos de tecla era secuencial, lo que se llama **modo crudo** (*raw mode*).

En los teclados actuales esto no ocurre así, ya que disponen de un microcontrolador y un buffer interno. El envío solo se hace al pulsar la tecla INTRO, hasta entonces es el propio teclado el que ejecuta las acciones de borrado. Es el **modo elaborado**, el que usamos por defecto.

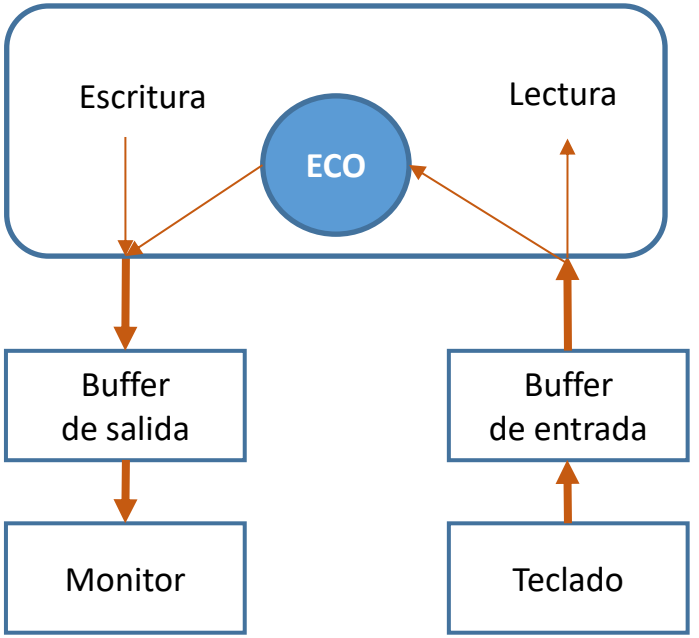


Códigos de tecla en un teclado QWERTY con disposición para inglés británico

Dispositivos de Entrada/Salida

El terminal

Cada vez que pulsamos una tecla en una línea de comandos esperamos ver el carácter correspondiente en el terminal. Esto no es automático, ni necesario para el funcionamiento del ordenador sino para el usuario, que necesita una realimentación visual. La aplicación terminal es la que se encarga de mantener esta ficción, pero puede anularse por programa. Hay situaciones en las que resulta imprescindible, como al escribir una clave. En otras, lo natural es que no haya eco. En un videojuego pulsar una tecla determinada puede desencadenar una acción de un personaje, resultaría muy molesto que el carácter apareciese en pantalla.



Dispositivos de Entrada/Salida

Almacenamiento secundario

El almacenamiento secundario es el que persiste aunque se interrumpa la alimentación. En este punto, los cursos de Sistemas Operativos dedican un extenso apartado a las características físicas del disco electromagnético, un dispositivo que ha sido una obra cumbre de la ingeniería tecnológica pero que está en fase de obsolescencia terminal.

Por esta razón, este curso ya no estudiamos los discos magnéticos, ni los algoritmos para optimizar el posicionamiento de la cabeza lectoescritora. En su lugar, nos centraremos en la alternativa, los dispositivos de estado sólido, que se siguen llamando discos por costumbre aunque no tienen partes móviles.

Dispositivos de Entrada/Salida

Almacenamiento secundario

Los discos de estado sólido usan la tecnología de almacenamiento NAND Flash. No entraremos en los detalles tecnológicos, baste saber que la capacidad de almacenamiento con este sistema por unidad de superficie es muy superior a la de la memoria SDRAM y no necesita refresco. A cambio, el tiempo de acceso es mayor.

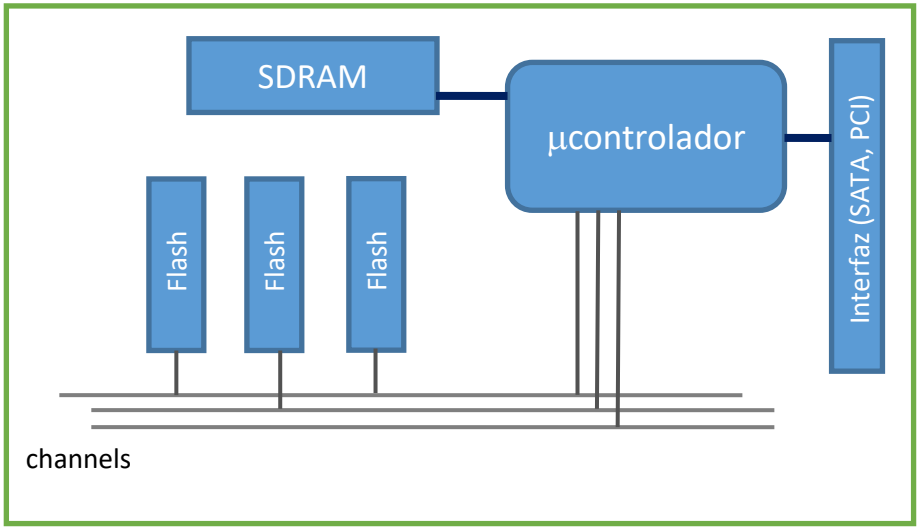
Como los dispositivos SSD llegaron para sustituir a los magnéticos, las primeras versiones heredaron algunas características de funcionamiento simuladas para resultar compatibles. Aquí podemos citar el sector de 512 bytes que en memoria flash no tiene ningún sentido físico o la interfaz SATA, cuya velocidad máxima de transferencia efectiva es del orden de 600 Mbyte/s.

En los últimos años se impone el factor de forma m.2 para los nuevos “discos” SSD, que permite velocidades de transferencia un orden de magnitud mayor. Esta es la solución en los nuevos equipos de sobremesa. En los smart pones, la memoria SSD está presente desde los inicios, estos dispositivos han sido los que han fomentado la innovación en esta tecnología que ha arrinconado a los discos magnéticos a los grandes sistemas redundantes tipo RAID en los CPDs.

Dispositivos de Entrada/Salida

Almacenamiento secundario

Un dispositivo SSD (ya sea un disco, una memoria USB o una tarjeta micro o nano SD) es un microordenador en sí mismo que realiza tareas muy complejas.



SSD

Dispositivos de Entrada/Salida

Almacenamiento secundario

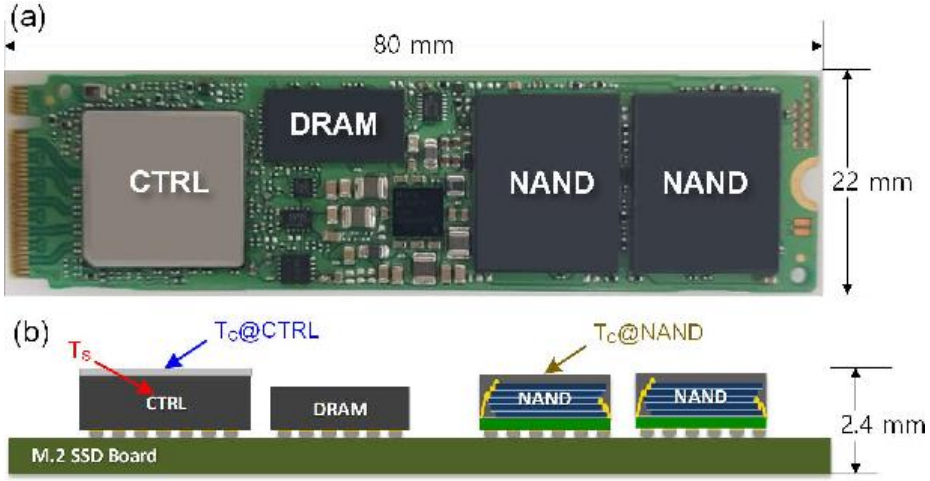
Los elementos internos de un SSD son:

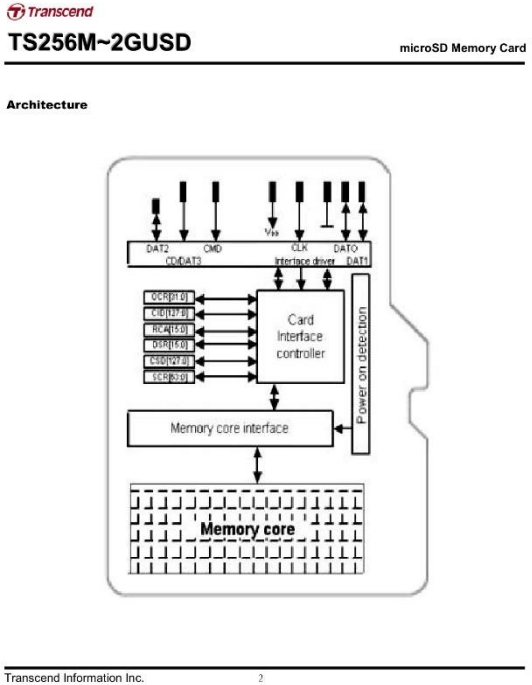
- Microcontrolador. Pequeña CPU que contiene y ejecuta la inteligencia del SSD.
- SDRAM. Banco de memoria auxiliar del microcontrolador que actúa también como caché para aumentar la velocidad de acceso.
- Interfaz. Realiza el diálogo con el driver, las más habituales son SATA y PCI.
- Bancos de memoria Flash que almacenan los datos.
- Channels. Buses internos de 8 bits que permiten el intercambio de información entre el microcontrolador y los bancos de memoria. Hay varios para permitir escrituras y lecturas en paralelo

El SSD tiene su propio sistema de ficheros (FFS: *Flash File System*) que es transparente al programador de sistemas. Es parecido a la FAT. La traducción de bloques a posiciones físicas de memoria RAM la realiza la *Flash Translation Layer*. Debido a que la vida media de una celda flash depende del número de accesos la FTL se ocupa de distribuir los ficheros por toda la memoria y no estresar unas pocas zonas. Este proceso que conlleva movimientos periódicos internos se llama *Wear Leveling*.

Dispositivos de Entrada/Salida

Almacenamiento secundario





Wang Y, Dong X, Zhang X, Wang L. Measurement and analysis of SSD reliability data based on accelerated endurance test. *Electronics*. 2019 Nov;8(11):1357.

Dispositivos de Entrada/Salida

Almacenamiento secundario

Por el mismo motivo, los “sectores” que quedan desocupados no se borran físicamente, simplemente se marcan como libres para poder ser reutilizados. El microcontrolador realiza también labores de compactación y borrado de bancos aislados con una rutina *garbage collector.* Al igual que en un disco magnético los sectores se van degradando con el tiempo y el controlador debe marcarlos como defectuosos, en el SSD existe un procedimiento similar.

Pregunta: ¿En qué consiste el timo de los USB de gran capacidad y bajo precio?

Dispositivos de Entrada/Salida

Almacenamiento secundario

Dice el refrán que nadie vende duros a cuatro pesetas. Si encuentras una oferta irresistible de 2TB por 5 dólares hay gato encerrado, seguro.

En el proceso de fabricación de las memorias flash un tercio de los circuitos son defectuosos. En lugar de destruirse se envían a reciclar, pero los compran revendedores sin escrúpulos.

El firmware del microcontrolador está programado para responder con la capacidad nominal (USB de 2 TB de capacidad) pero es posible que en el chip defectuoso solo haya 16 GB de bancos de memoria flash correctos. La impresión del usuario es que el USB funciona hasta que supera esa cantidad, momento en el cual pueden corromperse los ficheros y perder el trabajo almacenado. Este mismo problema puede ocurrir con tarjetas microSD para teléfonos o cámaras.

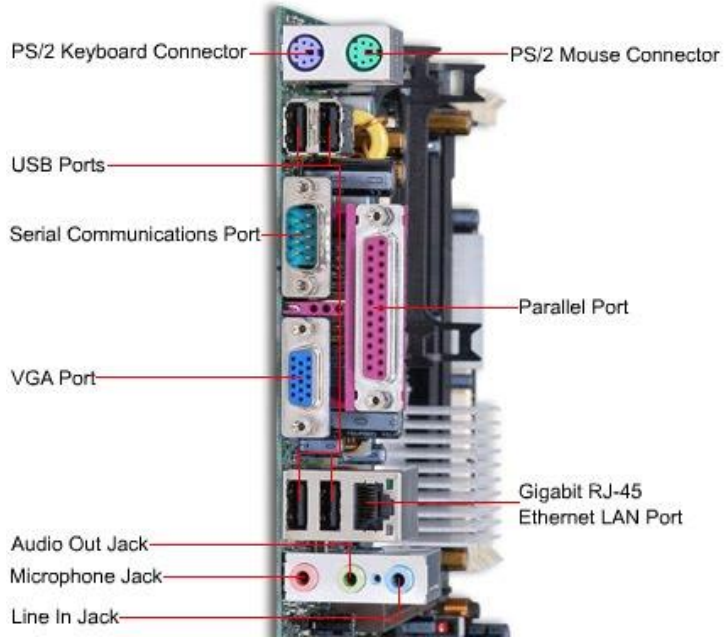
La herramienta RPMPrepUSB permite comprobar la capacidad efectiva de una memoria

Dispositivos de Entrada/Salida

USB

El Universal Serial Bus (USB) se ha convertido en la interfaz más versátil para conexión de todo tipo de dispositivos. La versión 1.0 nació en 1995, por acuerdo de los principales fabricantes de la época para simplificar la conexión de periféricos.

Debido al éxito alcanzado le siguieron la versión 2.0 en 2000, de amplia aplicación aun hoy en día, y la 3.0 en el año 2008. Para entonces USB ya había extendido su ámbito de aplicación a todo tipo de dispositivos electrónicos.



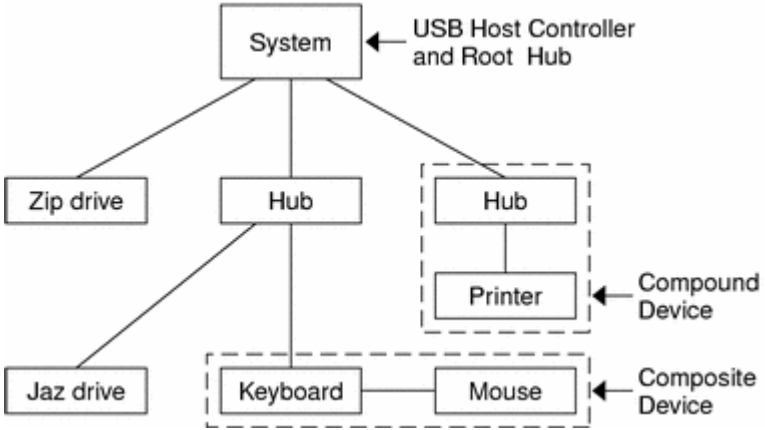
Backpanel con conectores *legacy*

Dispositivos de Entrada/Salida

USB

En la versión 2.0 USB utiliza dos hilos para la transmisión de datos, con código de línea NRZ y dos hilos para la alimentación. El modo de transmisión es semidúplex (half duplex) lo que quiere decir que pueden enviarse datos en ambos sentidos pero no de forma simultánea.

La topología de USB es un árbol, con un nodo raíz llamado *host* (habitualmente el PC) del que cuelgan el resto, llamados *slaves*. EN 2.0 puede haber 127 nodos esclavos por cada host. Un nodo esclavo del que cuelgan otros se denomina *hub*.



Topología USB

Dispositivos de Entrada/Salida

USB

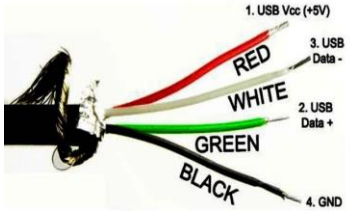
La alimentación en USB 2.0 es asimétrica, solo la proporciona el extremo que actúa como *hub*. Cada dispositivo se direcciona con un *end point*. El host es siempre el @1, el @0 está reservado.

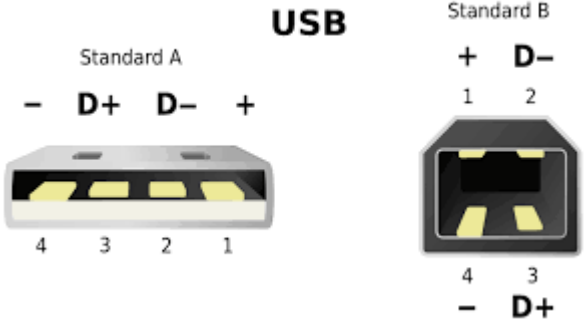
El gran éxito de USB se debe a la capacidad *plug-and-play* que permite conectar dispositivos sin apagar el PC. Esto es posible porque los SO modernos tienen la capacidad de cargar el driver adecuado de forma dinámica, una vez que se ha identificado el dispositivo, incluso buscando por internet.

USB proporciona 4 tipos de transferencia: Control (se usa durante la configuración), interrupción (para dispositivos como ratón o teclado), masiva (ficheros) e isócrona (audio/video). La comunicación entre el host y el dispositivo se realiza usando un protocolo con una trama especial llamada token. Solo el propietario del token puede escribir en el bus, evitándose de esta forma las colisiones. Todos están a la escucha, pero solo leen las tramas dirigidas a su *end point*.

Dispositivos de Entrada/Salida

USB

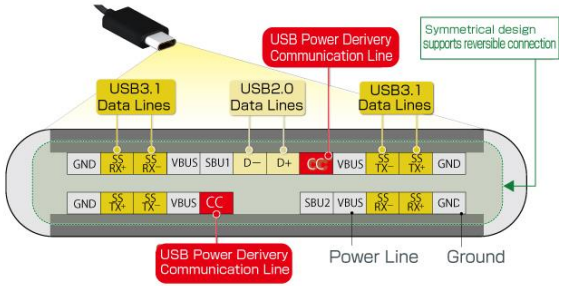




USB 2

USB 3.0 connector pinouts[50]

Pin	Color	Signal name		Description
		A connector	B connector	
Shell	N/A	Shield		Metal housing
1	Red	VBUS		Power
2	White	D−		USB 2.0 differential pair
3	Green	D+		
4	Black	GND		Ground for power return
5	Blue	StdA_SSRX−	StdB_SSTX−	SuperSpeed transmitter differential pair
6	Yellow	StdA_SSRX+	StdB_SSTX+	
7	N/A	GND_DRAIN		Ground for signal return
8	Purple	StdA_SSTX−	StdB_SSRX−	SuperSpeed receiver differential pair
9	Orange	StdA_SSTX+	StdB_SSRX+	



USB 3

Dispositivos de Entrada/Salida

USB

La propiedad más conocida de USB por el gran público es la capacidad de reconocer cualquier dispositivo al vuelo. Esta función exige un protocolo de sincronización (*handshacke*) entre el host y el dispositivo desconocido.

1. El host detecta la conexión de un nuevo dispositivo.
2. El host detecta si es de baja o alta velocidad.
3. El host resetea el dispositivo que toma el *end point* especial@0.
4. Se establece un canal (pipe) entre host y dispositivo. El host pide a este que envíe su descriptor.
5. El dispositivo responde con este descriptor que contiene datos como tipo de equipo, fabricante, versión, consumo, número de interfaces, etc.
6. Con esta información el host ya sabe qué driver usar o instalar si es la primera vez que se conecta.
7. El host asigna un número de *end point* libre al dispositivo. Un mismo dispositivo físico puede tener varios *end points*, como un modem USB que además tiene una memoria *flash*.

Dispositivos de Entrada/Salida

USB

El estándar USB 3.0 modificó de manera sustancial las posibilidades de estos equipos aunque manteniendo la compatibilidad con 2.0.

El conector es de 8 hilos, lo que permite que la comunicación sea full duplex (para los dispositivos 2.0 funciona con 4 hilos y half duplex). La alimentación es activa desde cualquiera de ambos extremos. Esto ha permitido que los PCs actuales puedan cargarse usando un puerto USB 3.0 conector tipo C, eliminando la conexión de carga propietaria. Prácticamente todos los móviles actuales soportan el mismo tipo de carga.

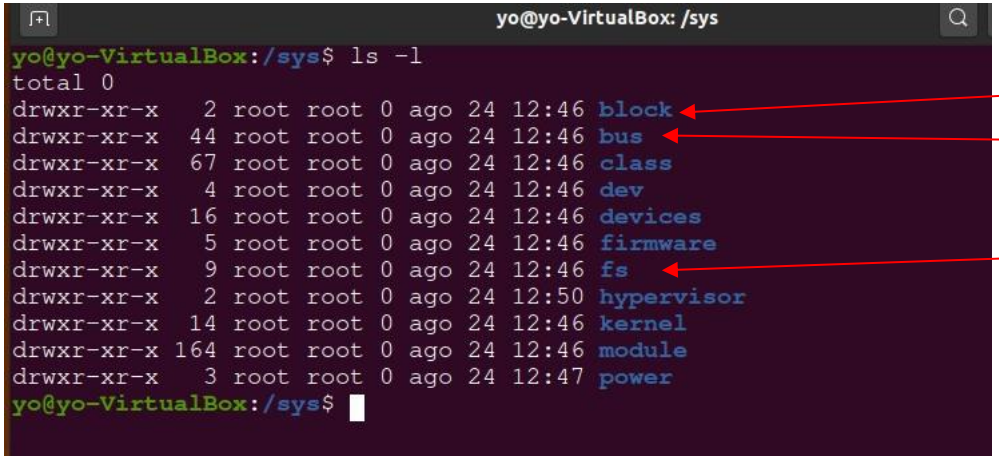
Con el tipo de transmisión llamado *super speed* se puede transmitir video en ambos sentidos. Esto ha permitido que los móviles puedan actuar como servidores de vídeo, conectados a un monitor sin necesidad de puerto microHDMI en el teléfono.

La velocidad de transferencia máxima en USB 3.2 gen 2x2 es de 20 Gbps, comparada con los 480 Mpbs de USB 2.

Dispositivos en Linux

¿Cómo podemos saber qué dispositivos tiene instalados nuestro equipo?

En Linux hay varias maneras. El sistema crea entradas en el directorio `/dev` por cada dispositivo sea físico o lógico (terminales). Otro directorio interesante en el que fijarse en `/sys`



Drivers en Linux

Los drivers son la capa del kernel que maneja los dispositivos. Hay dos formas de instanciarlos, compilándolos durante la generación del kernel o usando la capacidad *plug & play*.

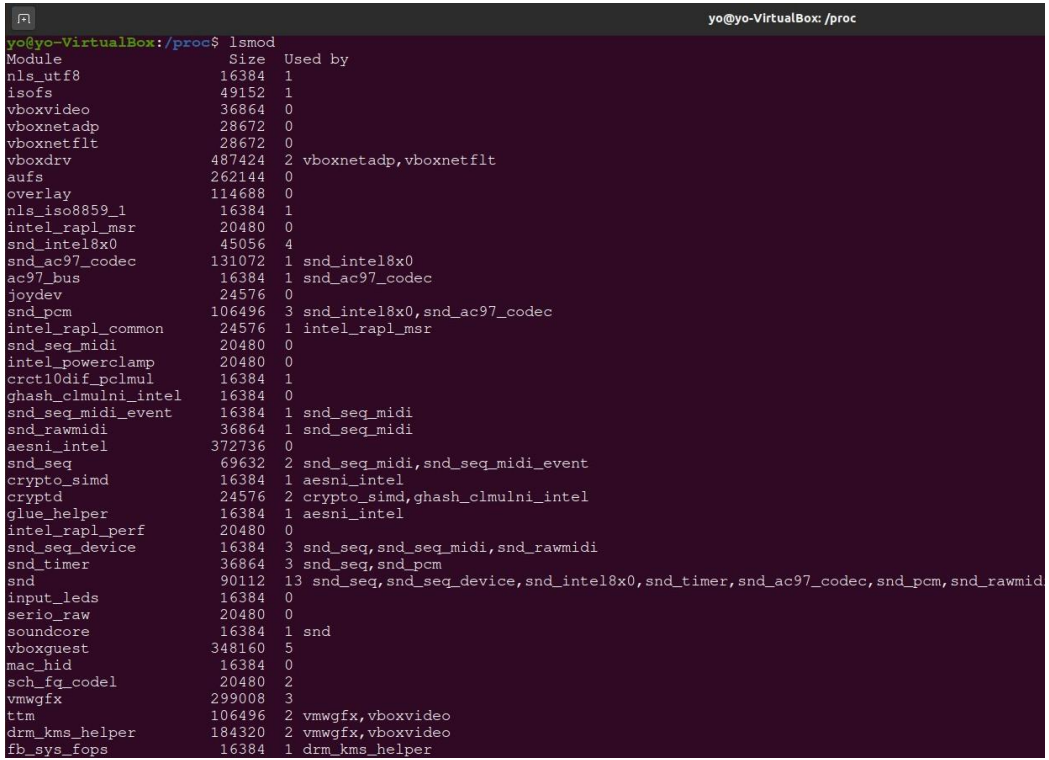
La primera manera es la que se usaba en los sistemas Unix primitivos. En aquellos tiempos el hardware era muy caro y no cambiaba a diario. El **sysadmin** configuraba a compilación para contemplar todos los equipos, generaba el instalable y arrancaba el sistema. Si, ocasionalmente, se compraba un disco adicional, se recompilaba el kernel. Esto no resultaría operativo en un mundo como el de los equipos de sobremesa, pero sigue teniendo sentido en los grandes CPD por motivos de seguridad. Si tenemos una infraestructura estable y no queremos que nadie pueda atacarla introduciendo virus por USB u otros elementos *plug & play*, este tipo de solución es muy robusta. Se inhabilita la opción plug & play por configuración.

Lo habitual es que los equipos personales tengan una gran cantidad de dispositivos que pueden conectarse, aunque no todos a la vez: discos de backup, teléfono móvil, Tablet, smart watch, equipos IoT… en este caso es imprescindible poder cargar drivers de forma dinámica. En la

Drivers en Linux

¿Cómo podemos saber qué drivers tiene cargados nuestro equipo?

En el directorio `/sys` hay un subdirectorio llamado module con una entrada por cada módulo. El comando `lsmod` vuelca una información más detallada.



Drivers en Linux

Los dispositivos Linux (Unix en general) se identifican por cifras llamadas *major* y *minor*. El número *major* identifica el tipo de dispositivo y sirve para saber qué driver tiene que manejarlo. El *minor* distingue los dispositivos de la misma clase. Puede verse esta información listando el directorio /dev. Cada dispositivo tiene asignado un inodo que almacena información sobre si es de tipo bloque o de tipo carácter y los números major y minor.

```
yo@yo-VirtualBox:/$ ls -ll /dev | grep sd
brw-rw----  1 root    disk         8,   0 ago 24 12:47 sda
brw-rw----  1 root    disk         8,   1 ago 24 12:47 sda1
brw-rw----  1 root    disk         8,   2 ago 24 12:47 sda2
brw-rw----  1 root    disk         8,   5 ago 24 12:47 sda5
yo@yo-VirtualBox:/$
```

Dispositivo tipo bloque

Dispositivo de bloque (disco SATA)

particiones

```
yo@yo-VirtualBox:/$ ls -ll /dev | grep ttyS
crw-rw----  1 root    dialout      4,  64 ago 24 12:47 ttyS0
crw-rw----  1 root    dialout      4,  65 ago 24 12:47 ttyS1
crw-rw----  1 root    dialout      4,  74 ago 24 12:47 ttyS10
crw-rw----  1 root    dialout      4,  75 ago 24 12:47 ttyS11
crw-rw----  1 root    dialout      4,  76 ago 24 12:47 ttyS12
crw-rw----  1 root    dialout      4,  77 ago 24 12:47 ttyS13
crw-rw----  1 root    dialout      4,  78 ago 24 12:47 ttyS14
crw-rw----  1 root    dialout      4,  79 ago 24 12:47 ttyS15
crw-rw----  1 root    dialout      4,  80 ago 24 12:47 ttyS16
crw-rw----  1 root    dialout      4,  81 ago 24 12:47 ttyS17
crw-rw----  1 root    dialout      4,  82 ago 24 12:47 ttyS18
crw-rw----  1 root    dialout      4,  83 ago 24 12:47 ttyS19
```

Dispositivo tipo carácter

Terminal

tty es la abreviatura de teletype

Drivers en Linux

Principios de diseño de un driver Linux:

- **API unificada**. Los dispositivos tienen que responder a los servicios `open()`, `close()`, `read()`, `write()` y `seek()`. El resto del diálogo driver-dispositivo se realiza con la función comodín `ioctl()`, por ejemplo para enviar comandos iniciales de configuración.

- **Nombrado coherente.** Los dispositivos generan entradas en el directorio /dev y los nombres tienen que seguir una norma de nombrado coherente. Hemos visto como en Linux los discos SATA se llaman /dev/sda, en BSD se llaman /dev/ada.

- **Buffering**. Los drivers actúan como memoria tampón para adecuar las velocidades de transferencias, para ello tienen que implementar los buffers necesarios.

- **Protección**. Los drivers tienen que soportar el mecanismo de control de acceso estándar para evitar que cualquier proceso de usuario pueda acceder a cualquier dispositivo físico.

Drivers en Linux

Los drivers de UNIX no entienden de formato de la información, solo ven flujos de bytes desde la memoria hacia los registros del dispositivo y viceversa. Los dispositivos se abren con el mismo servicio `open()` que los ficheros de disco.

```
int open(const char *path, int flags, mode_t mode)
```

La función devuelve el descriptor que se usará para todas las operaciones mientras permanezca abierto o -1 si hay error, exactamente igual que con un fichero. El cierre, como el lector habrá descubierto se hace con `close()`, mientras que las primitivas `read()` y `write()` realizan la transferencia de información.

Esta solución tan elegante permite redireccionar la entrada/salida de un proceso a cualquier fichero o dispositivo sin hacer cambios en el código. Cuando escribimos `ls -l > info.txt` el intérprete de comandos redirige la información que iba a aparecer por `stdout` (descriptor 1) por el descriptor correspondiente al fichero `info.txt`. De la misma forma, cuando conectamos la salida de un proceso a la entrada del siguiente, lo hacemos redireccionando la salida estándar del primero (descriptor 1) a la entrada estándar del segundo (descriptor 0).

Drivers en Linux

Hello kernel!

El desarrollo de módulos para el kernel tiene una serie de limitaciones que no aparecen en los procesos de usuario. Por ejemplo, no puede cargarse una librería como la `libc`, todo el código tiene que estar en el espacio kernel. Así, no podemos usar `printf()` para mostrar mensajes o `malloc()` para pedir memoria dinámica, pero existen funciones kernel equivalentes.

El desarrollo se hace en C convencional, aunque el listado de código kernel te parecerá un poco extraño al principio. Por ejemplo, es común encontrar ensamblador inline, es decir fragmentos de ensamblador dentro de una función escrita en C, es una posibilidad que ofrece `gcc` pero muy poco común en los procesos de usuario.

Tampoco existe la función `main()`, hay un entry point al módulo que se invoca con la función `module_init()`. Tampoco hay una función `exit()`, pero sí una `module_exit()` que se ejecuta cuando el módulo se descarga de memoria.

Drivers en Linux

Nuestro primer módulo kernel para Linux.

```
// Headers necesarios para compilar un módulo. Fíjate que no
// incluimos stdio.h porque no podemos usar la entrada/salida estándar

#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("GPL");

// La sintaxis de la función es convencional
int hola_init(void)
{
    printk(KERN_INFO “Hola kernel!!!\n");  //Equivalente a printf()
                                           //Fíjate en que no hay coma
    return 0;                              //KERN_INFO es una macro
}
void hola_exit(void)
{
    printk(KERN_INFO “Adios kernel\n");
}
module_init(hola_init);  //Se ejecuta al cargar el módulo
module_exit(hola_exit);
```

Drivers en Linux

Hello kernel!

Un módulo no puede compilarse como un proceso normal con un simple `gcc -o`. Para poder generar el código hace falta un `Makefile` con este contenido

```
ifneq ($(KERNELRELEASE),)
    obj-m := hello.o

else
    KERN_DIR ?= /usr/src/linux-headers-$(shell uname -r)/
    PWD := $(shell pwd)

all:
	$(MAKE) -C $(KERN_DIR) M=$(PWD) modules

endif
```

Salva el fichero en el mismo directorio que `hello.c` y ejecuta el comando `make` con `sudo`.

Carga el módulo con el comando sudo `insmod hello.ko`

Drivers en Linux

Hello kernel!

```
yo@yo-VirtualBox: ~/probatinas
yo@yo-VirtualBox:~/probatinas$ sudo make
make -C /usr/src/linux-headers-5.4.0-39-generic/ M=/home/yo/probatinas modules
make[1]: se entra en el directorio '/usr/src/linux-headers-5.4.0-39-generic'
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /home/yo/probatinas/hello.o
see include/linux/module.h for more information
make[1]: se sale del directorio '/usr/src/linux-headers-5.4.0-39-generic'
yo@yo-VirtualBox:~/probatinas$ sudo insmod hello.ko
yo@yo-VirtualBox:~/probatinas$
```

Drivers en Linux

Hello kernel!

El módulo funciona perfectamente, pero hay que cambiar nuestra manera de pensar al trabajar con el kernel. Lo primero es comprobar si está cargado y para eso ya sabemos qué comando hay que emplear: `lsmod`

```
yo@yo-VirtualBox:~/probatinas$ sudo lsmod
Module                  Size  Used by
hello                  16384  0
nls_utf8               16384  1
isofs                  49152  1
vboxvideo              36864  0
vboxnetadp             28672  0
vboxnetflt             28672  0
vboxdrv               487424  2 vboxnetadp,vboxnetflt
aufs                  262144  0
overlay               114688  0
nls_iso8859_1          16384  1
intel_rapl_msr         20480  0
intel_rapl_common      24576  1 intel_rapl_msr
intel_powerclamp       20480  0
crct10dif_pclmul       16384  1
ghash_clmulni_intel    16384  0
joydev                 24576  0
aesni_intel           372736  0
crypto_simd            16384  1 aesni_intel
cryptd                 24576  2 crypto_simd,ghash_clmulni_intel
glue_helper            16384  1 aesni_intel
snd_intel8x0           45056  2
intel_rapl_perf        20480  0
```

Drivers en Linux

Hello kernel!

El kernel no tiene un terminal asociado, no escribe por la pantalla sino que graba sus mensajes en `/var/log/kern.log`

Cada vez que cargamos `hello` con `insmod` aparecerá el mensaje de entrada y cada vez que lo descargamos con `rmmod`, el de despedida. Es sencillo.

```
yo@yo-VirtualBox:~/probatinas$ sudo tail /var/log/kern.log
Aug 25 15:00:05 yo-VirtualBox kernel: [   45.062957] rfkill: input handler disab
led
Aug 25 15:00:08 yo-VirtualBox kernel: [   62.178363] rfkill: input handler enabl
ed
Aug 25 15:00:19 yo-VirtualBox kernel: [   73.095688] ISO 9660 Extensions: Micros
oft Joliet Level 3
Aug 25 15:00:19 yo-VirtualBox kernel: [   73.105977] ISO 9660 Extensions: RRIP_1
991A
Aug 25 15:00:20 yo-VirtualBox kernel: [   74.185681] rfkill: input handler disab
led
Aug 25 16:13:32 yo-VirtualBox kernel: [ 4466.670575] hello: module license 'unsp
ecified' taints kernel.
Aug 25 16:13:32 yo-VirtualBox kernel: [ 4466.670576] Disabling lock debugging du
e to kernel taint
Aug 25 16:13:32 yo-VirtualBox kernel: [ 4466.684139] Hola kernel!!!
Aug 25 16:20:17 yo-VirtualBox kernel: [ 4871.546507] Adios kernel
Aug 25 16:20:33 yo-VirtualBox kernel: [ 4887.506355] Hola kernel!!!
yo@yo-VirtualBox:~/probatinas$
```