

Sistemas Operativos

2. Conceptos básicos

Javier García Algarra

javier.algarra@u-tad.com

Miguel Ángel Mesas

miguel.mesas@u-tad.com

2021-2022

Introducción

En esta lección vamos a describir de forma resumida lo que es un Sistema Operativo y sus componentes. Es necesario tener una visión global antes de pasar a estudiar en detalle cada componente. Por ejemplo, no puede entenderse lo que es la memoria virtual (lección 4) sin haber entendido antes la gestión de procesos (lección 3), pero tampoco es posible comprender por qué un cambio de contexto (lección 3) resulta tan costoso en tiempo por la necesidad de recargar páginas de memoria dinámica (lección 4) o las cachés (lección 4).

Los sistemas operativos tienen una fuerte relación con el hardware, puesto que todas las interacciones programa-dispositivo pasan obligatoriamente por una llamada. En esta lección repasaremos también la arquitectura de los microprocesadores Intel x86 y los compararemos con los de arquitectura ARM cuyo ámbito de aplicación es cada día más extenso.

Conceptos

Un Sistema Operativo es el programa más importante que ejecuta el ordenador y tiene dos misiones fundamentales: **gestionar los recursos** de manera que los programas vean una máquina completa a su disposición y no interfieran entre ellos y **abstraer los detalles de los dispositivos hardware** para facilitar la labor del programador.

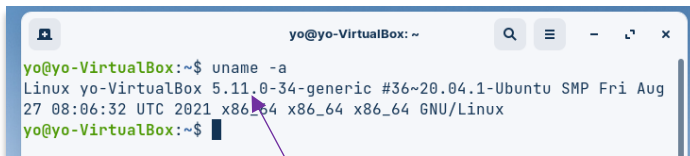
El Sistema Operativo crea y gestiona procesos. Un **proceso** es una instancia ejecutable de un programa y requiere código, datos y una serie de **metadatos** con los que el Sistema Operativo realiza su gestión. Dos instancias de un mismo ejecutable (por ejemplo, dos “Bloc de Notas”) son procesos distintos.

Todas las operaciones de entrada/salida pasan forzosamente por el Sistema Operativo. El sistema ofrece una **API normalizada** hacia los programas de usuario para poder invocar sus servicios.

Conceptos

La parte más importante del SO se llama **kernel** (núcleo). Se ejecuta en un modo privilegiado con acceso a toda la memoria, recursos y registros de la CPU. Los procesos de usuario no pueden usar todo el juego de instrucciones ni ven algunos registros del microprocesador.

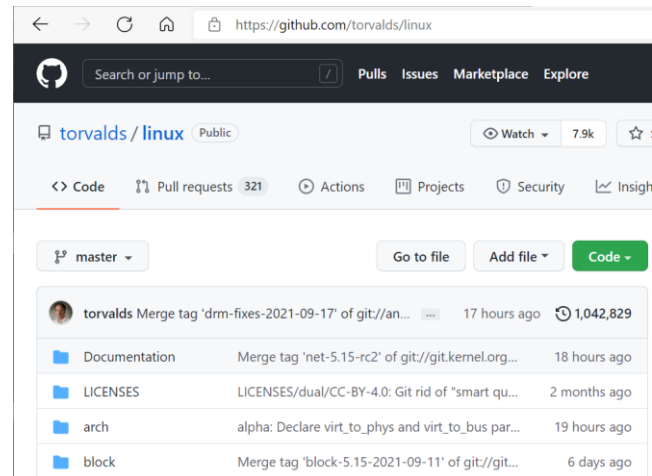
Repo en github



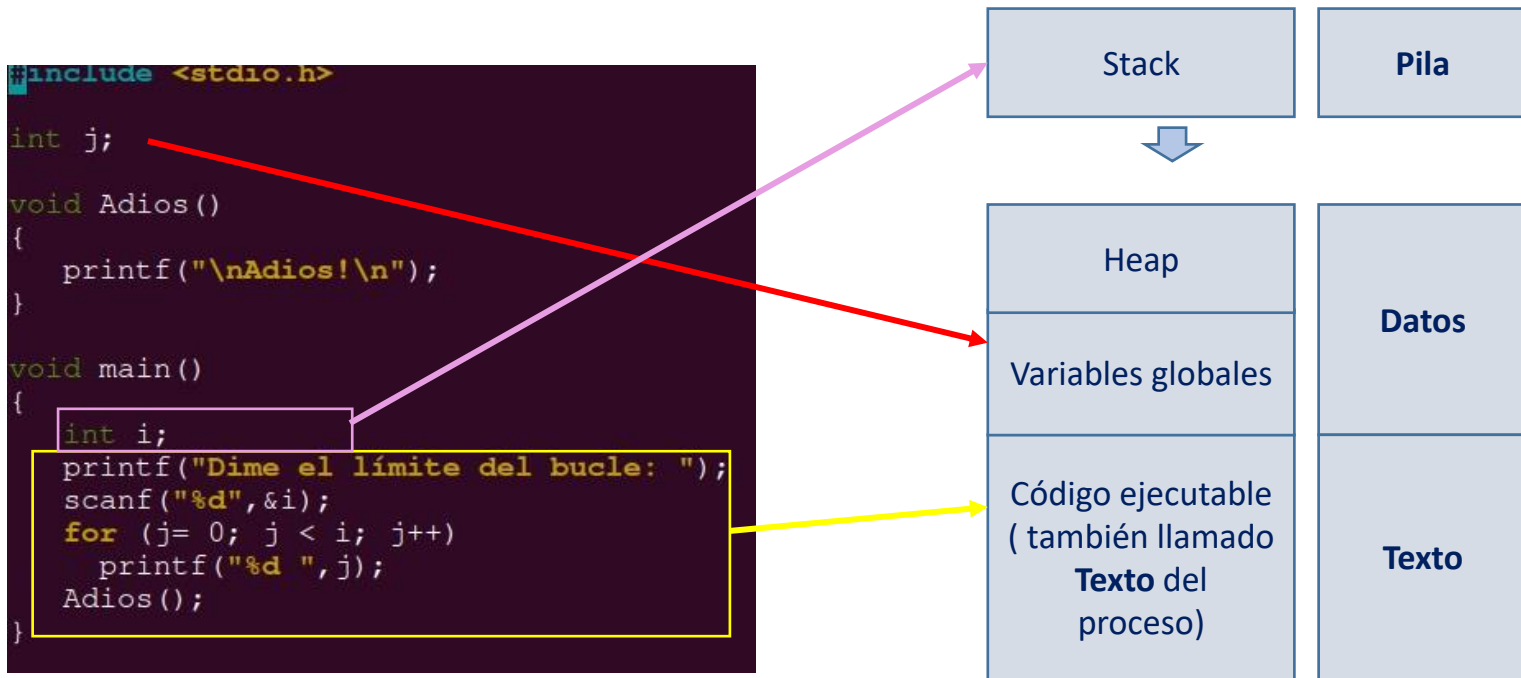
```
yo@yo-VirtualBox: ~  
yo@yo-VirtualBox:~$ uname -a  
Linux yo-VirtualBox 5.11.0-34-generic #36~20.04.1-Ubuntu SMP Fri Aug  
27 08:06:32 UTC 2021 x86_64 x86_64 x86_64 GNU/Linux  
yo@yo-VirtualBox:~$
```

A terminal window titled 'yo@yo-VirtualBox: ~' showing the output of the 'uname -a' command. The output displays system information including the kernel version '5.11.0-34-generic'. A purple arrow points from the text 'Versión del kernel de la distribución que estamos usando' to the kernel version string in the terminal output.

Versión del kernel de la
distribución que estamos usando



Proceso



Un **proceso** tiene código máquina ejecutable (llamado Texto), datos estáticos y funciones que se llaman usando el stack. Los parámetros y variables de las funciones se instancian también en el stack. El heap se usa para el manejo de la memoria dinámica. La zona de texto y variables globales es de tamaño fijo, el stack y el heap, de tamaño variable.

Proceso

La unidad mínima de ejecución en una CPU es la instrucción de código máquina. Una vez que la Unidad de Control comienza su decodificación la instrucción tiene que completarse. Como consecuencia, solo un proceso puede estar en ejecución en cada instante para un mismo núcleo.

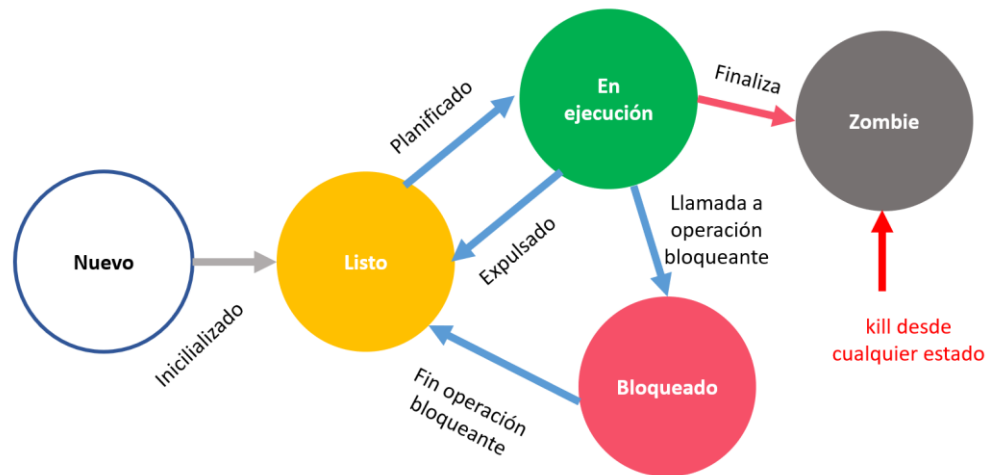
Como los procesos no están activos necesariamente todo el tiempo, el Sistema Operativo puede ir concediendo la CPU por turnos, de forma que da la sensación de que los procesos corren en paralelo, aunque siempre lo hacen secuencialmente.

Para que esto pueda ocurrir el Sistema Operativo tiene que ocupar la CPU en algún periodo. Esto ocurre cada vez que hay una interrupción, el proceso es desalojado y se ejecuta el kernel, que puede decidir conceder la CPU a otro proceso al terminar.



Ejecución paralela de cuatro procesos en una CPU

Proceso



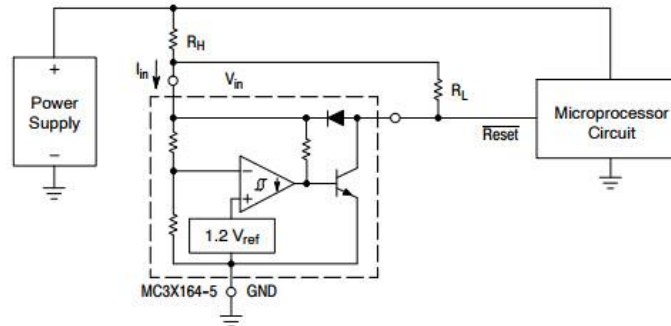
El ciclo de vida de un proceso se puede representar mediante una máquina de estados. En cada instante solo hay un proceso en ejecución. Todos los demás que podrían estar ocupando la CPU se encuentran en una cola, en el estado Listo. El proceso en ejecución se bloquea (también se dice que “duerme”) cuando invoca una operación de entrada/salida de duración imprevisible, como una lectura desde teclado. Queda en ese estado, sin consumir CPU hasta que se produce el evento y el Sistema Operativo lo pasa a la cola de procesos listos para ejecución.

Arranque del sistema operativo

Desde que ponemos en marcha un equipo hasta que el sistema está plenamente operativo sucede una secuencia compleja de eventos.

1 Reset

Cuando el ordenador se enciende, el microprocesador recibe una señal por el pin de RESET. El efecto de esta señal es dar a todos los registros un valor inicial, incluyendo el contador de programa. Así se consigue que la CPU empiece a ejecutar código máquina, siempre el situado en esa dirección especial, donde normalmente habrá un salto hacia la rutina que ejecutará el siguiente paso. En los procesadores Intel de 64 bits esa dirección es **0xFFFFF0**, en ARM **0xFFFF0000**.



Arranque del sistema operativo

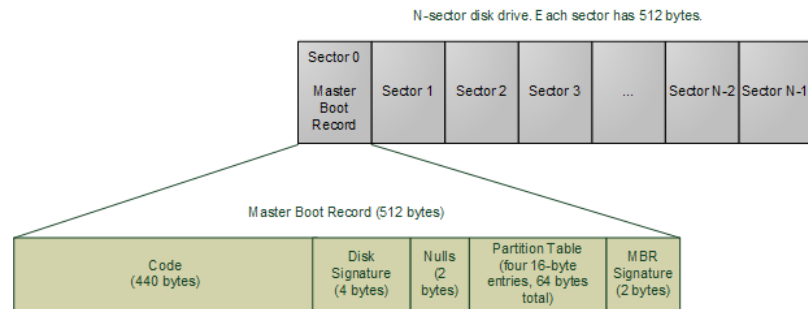
2 Iniciador hardware

El equipo saltará a un pequeño programa grabado en ROM o Flash proporcionado por el fabricante con el equipo (BIOS/UEFI en un PC). Este programa procede a detectar los dispositivos conectados y realiza unas **pruebas** mínimas de funcionamiento. En inglés se conoce como **POST** (Power-on Self-Test). Al terminar con éxito el POST, este firmware busca un dispositivo de almacenamiento masivo (disco, USB) que tenga un sector de arranque configurado. Para ello sigue el orden especificado en la BIOS. Hasta este momento, toda la ejecución es en modo supervisor y con memoria real.

El iniciador hardware carga en memoria el contenido del **MBR** (Master Boot Record) de la unidad de arranque y cede el control. El MBR ocupa el primer sector y contiene tan solo 512 bytes de código. En esos 512 bytes puede haber un salto hacia el cargador del sistema (Windows, Linux, Android) o un completo selector de arranque como LILO o GRUB, que tienen una pantalla de entrada que permite al usuario elegir qué sistema arrancar si hay varios instalados.

El código contenido en el MBR tampoco es parte del sistema operativo, es un programa autosuficiente.

Arranque del sistema operativo



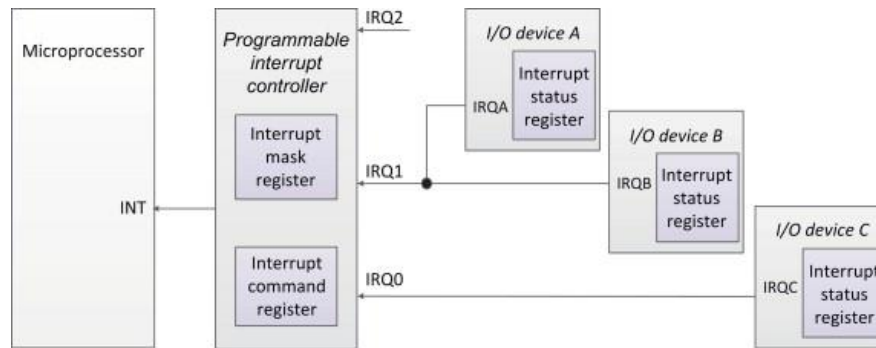
3 Carga del Sistema Operativo

A continuación comienza la carga en memoria del **Sistema Operativo residente**, la parte que está siempre residente (kernel, algunos drivers). Todavía estamos en modo supervisor y con memoria real. El SO residente crea sus tablas de gestión (BCP, páginas de memoria, ficheros, etc...) y cuando las concluye con éxito habilita las interrupciones y crea un proceso especial llamado **init** en Unix. Este proceso es muy especial porque es el antepasado de todos los procesos del sistema. En Unix todo proceso desciende de otro. El proceso **init** lee los scripts almacenados en `/etc/init.d` para ir creando todos los demás que se arrancan en forma automática. Todos ellos se ejecutan ya en modo usuario. En versiones más recientes de Linux **init** ha sido sustituido por **systemd**.

Activación del Sistema Operativo

El Sistema Operativo debe ocupar la CPU el mínimo tiempo posible pero, por otra parte, tiene que ejecutarse de forma frecuente para poder cumplir su cometido. El kernel (núcleo) está cargado en RAM de forma permanente para evitar el enorme coste que supondría leer su código desde disco cada vez. Mientras se ejecuta un proceso de usuario, no puede acceder al espacio de direcciones del kernel, que por definición está protegido salvo para el modo usuario.

La forma de despertar al kernel es siempre mediante una interrupción. Esta es una señal eléctrica que genera un dispositivo externo (un teclado, un disco, el adaptador WiFi) y que sucede en cualquier instante. Cuando ocurre una interrupción la CPU lo sabe porque recibe una señal por el pin INTR o el NMI (Non-Maskable Interrupt). También puede producirse por una excepción (división por 0) o por una instrucción especial de código máquina.



Activación del Sistema Operativo

Para procesar la interrupción hay que poner en el contador de programa la dirección del código de atención, pero entonces perdemos su contenido que apuntaba a la siguiente instrucción del programa interrumpido. La propia rutina de atención no tiene ya la posibilidad de recuperar ese valor.

¿Cómo se soluciona esta paradoja?



Activación del Sistema Operativo

Es la CPU la que salva, como mínimo, el contador de programa y el registro de flags en el stack como parte de la instrucción de la instrucción de interrupción y los recupera en el retorno. No debería sorprendernos porque es muy parecido a lo que ocurre con una llamada a subrutina en código máquina (CALL en el ensamblador del Z80). La dirección de retorno se guarda en el stack y se recupera por el hardware.

Salvados por la rutina de atención a la interrupción



Stack del usuario

Salvados por la CPU al pasar a modo interrupción

Activación del Sistema Operativo

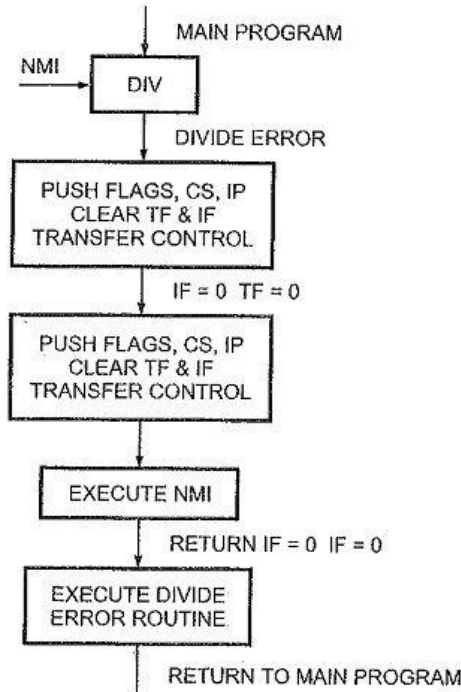


Fig. 9.4 Flow-chart for divide error routine

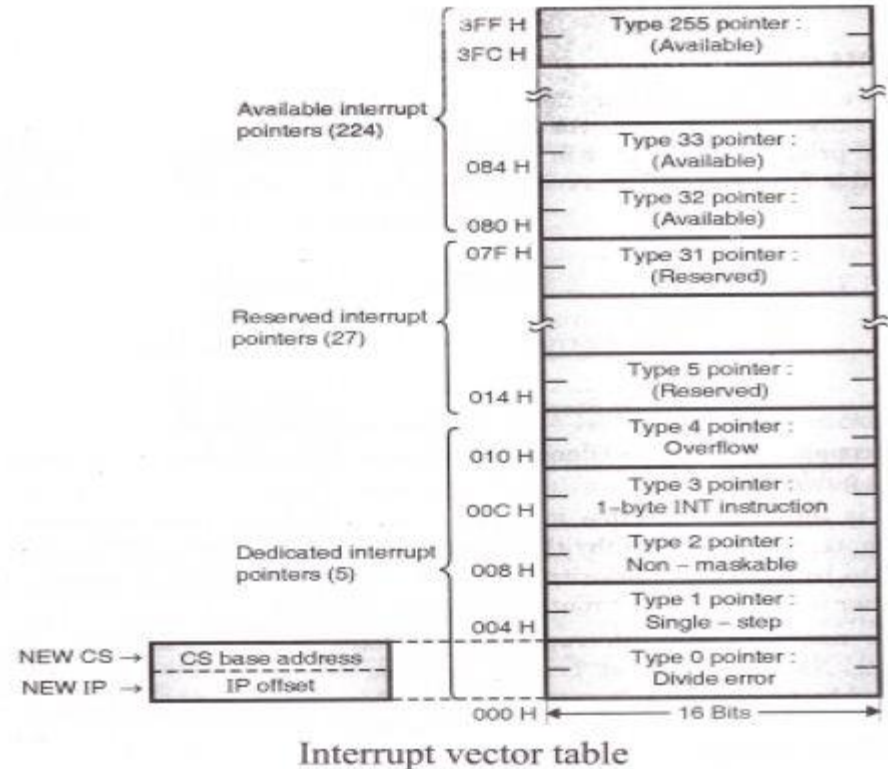
La CPU termina de ejecutar la instrucción de código máquina en curso, salva el PC y el registro de Flags del proceso en su stack y carga en el contador de programa una dirección predefinida donde se almacena la rutina de atención de interrupciones, que es parte del kernel. Esta operación pone la CPU en modo supervisor.

La rutina salva el estado del procesador en el stack de usuario, reconoce qué dispositivo es el que solicita atención, y con esa información localiza en la tabla de vectores de interrupción dónde está el código concreto (por ejemplo, atender USB). Cuando finaliza la atención, la instrucción de código máquina retorno de interrupción, recupera los valores de los registros del procesador en el momento de la interrupción devuelve la CPU a modo usuario.

Activación del Sistema Operativo

Tabla de vectores de interrupción en el 8086. Los valores más bajos son para atender las excepciones y la interrupción SW. A continuación había un rango de interrupciones predefinidas para la atención de los circuitos de servicio del 8086 y quedaban 224 libres para poder manejar otros dispositivos de entrada salida.

Las tablas de los procesadores actuales son mucho más complejas.



Activación del Sistema Operativo

Hay dos interrupciones que desempeñan un papel importante en el funcionamiento del Sistema. La **interrupción periódica *tick*** (de 1 a 15 ms, depende del sistema) es una señal eléctrica generada a partir del reloj de la CPU mediante un divisor de frecuencia. Esta interrupción garantiza que el kernel despierta a menudo para poder llevar a cabo la función de planificación, que consiste en decidir si el proceso actual sigue ocupando la CPU y, si no, cual debe sustituirlo.

Tanto el tick como las **interrupciones hardware** despiertan el kernel sin intervención del programador. Hemos afirmado que toda la entrada/salida pasa por él, pero no se puede llamar a una función del kernel desde el usuario porque su espacio de direcciones es inaccesible.

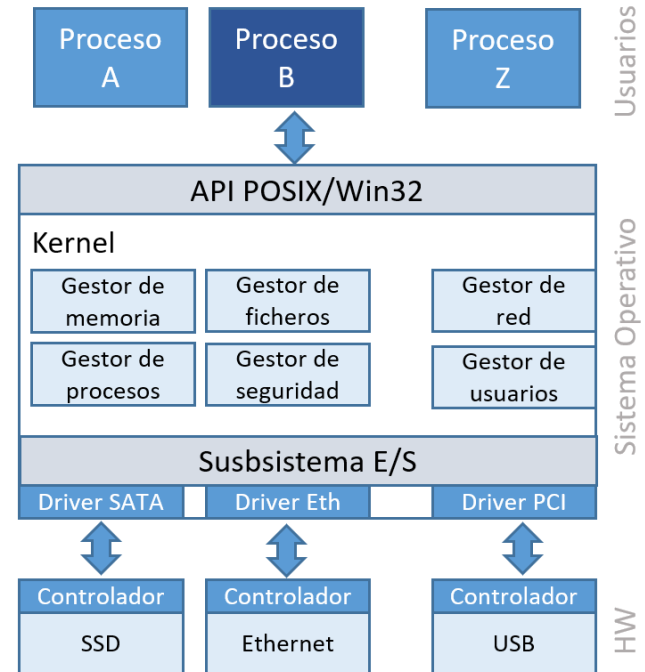
La solución es **una interrupción SW (INT en x86)**, una instrucción de código máquina que pone a la CPU en modo interrupción y tiene una entrada en la tabla de vectores de atención como otra cualquiera. La forma de invocar un servicio es colocar en el stack el código de la operación que queremos que realice el sistema, despertarlo con una interrupción SW y recibir de vuelta el valor en el stack.

Todos los sistemas ofrecen bibliotecas wrapper, que envuelven estas llamadas con una función convencional y abstraen al usuario.

Activación del Sistema Operativo

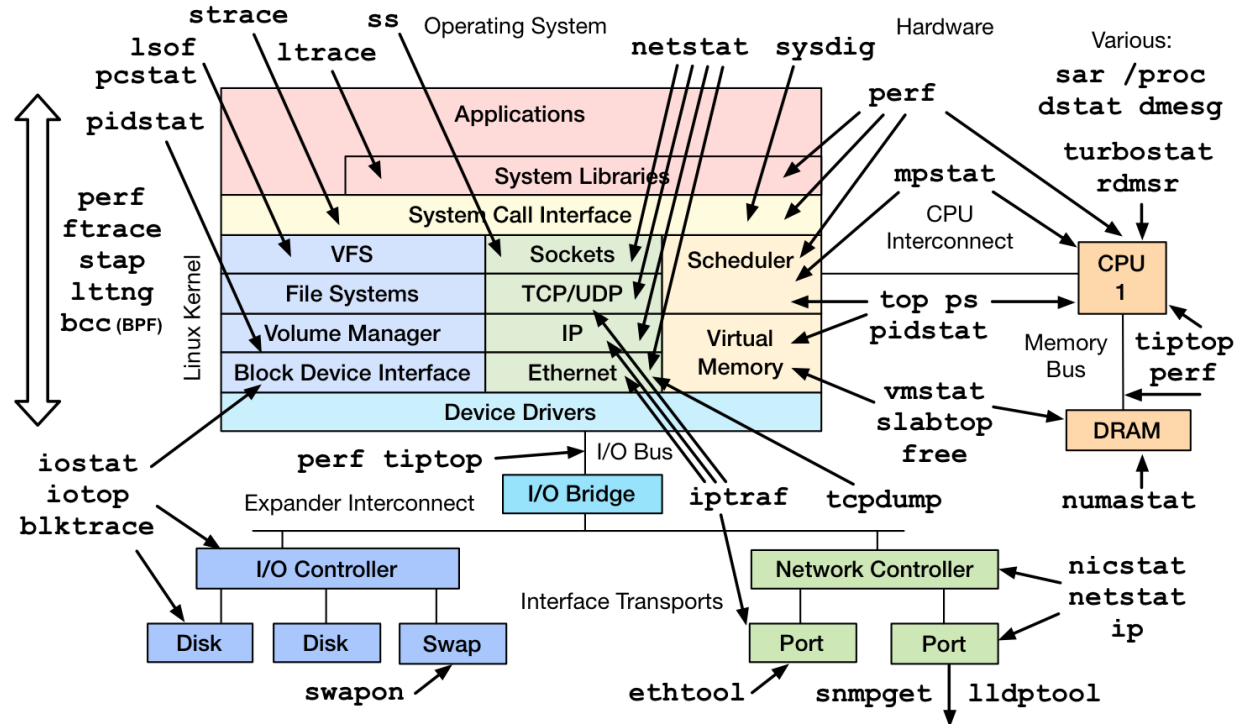
La sintaxis de estas funciones ha terminado cristalizando en estándares de facto como POSIX para los sistemas Unix o Win32/Win64 para Windows.

POSIX es una colección de estándares del IEEE que definen la API que debe ofrecer el sistema, no indican cual debe ser la implementación concreta. Los sistemas Unix ofrecen interfaz POSIX a los programadores. Las últimas versiones del kernel de Windows también tienen interfaz POSIX además de la nativa de Windows. Eso permite construir emulaciones muy completas de la shell como CygWin o MinGW.



Comandos de traza en Linux

Linux Performance Observability Tools



Gestión de memoria

Mantener un sistema con decenas o centenares de procesos cargados, con llegada y salida continua de nuevos procesos o threads no sería posible si todos tuvieran que tener el mapa de memoria cargado completamente en la RAM.

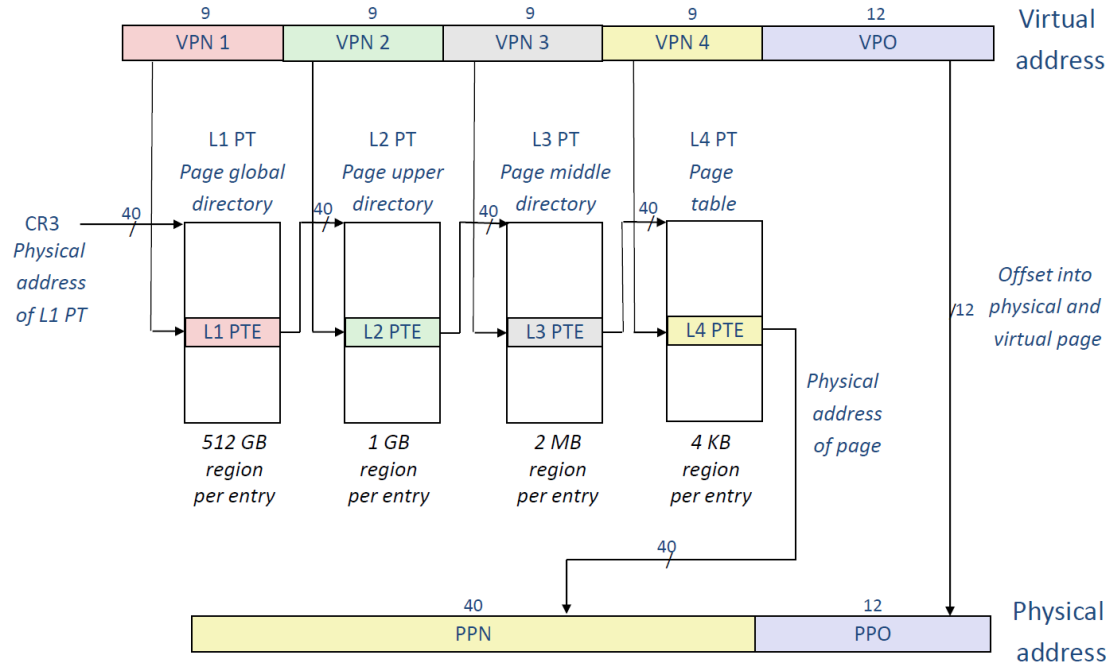
Una combinación de circuitos de la propia CPU (TLB, caché) permite tener cargados solo los fragmentos de código en ejecución a corto plazo mientras que el resto se va leyendo desde o escribiendo a disco cuando no se usa. Esta complejísima gestión la realiza el kernel, y la solución más común que detallaremos en la lección 4 es la memoria virtual con paginación. El Sistema Operativo oculta al programador toda esta complejidad pero también es capaz de ofrecerle gestión de memoria dinámica o creación de segmentos de memoria compartida para comunicación entre procesos.

Además, el equipo tiene que supervisar de forma continua (por hardware) que ningún proceso accede a direcciones indebidas (datos de otros procesos, datos del kernel, su propio código en modo escritura). Si esto ocurre, tiene que atender de forma adecuada las interrupciones que causan esas excepciones.

Gestión de memoria

Intel Core i7

48 bits



52 bits

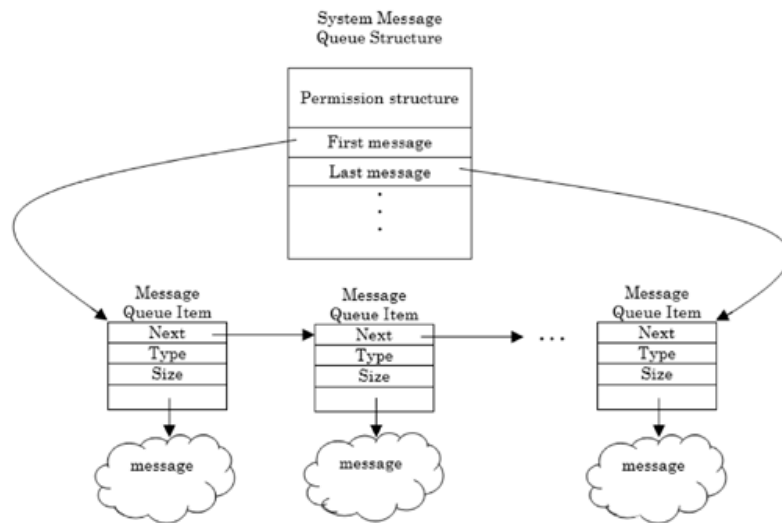
Esto es una tabla de páginas de memoria de 4 entradas para traducir direcciones virtuales de 48 bits a físicas de 52. Es un árbol cuyas hojas son estructuras encadenadas mediante punteros. Los Sistemas Operativos están escritos en C, sería conveniente repasar estos conceptos de Programación II

Comunicación y sincronización

Uno de los principios de diseño de los Sistemas Operativos modernos, como hemos visto ya, es la separación de espacios de memoria entre procesos por motivos de seguridad. Sin embargo, en numerosas ocasiones es necesario que los procesos colaboren entre sí y se comuniquen. Para ello, el Sistema Operativo ofrece una colección de mecanismos.

1. Señales
2. Pipes
3. Colas de mensajería
4. Sockets
5. Memoria compartida

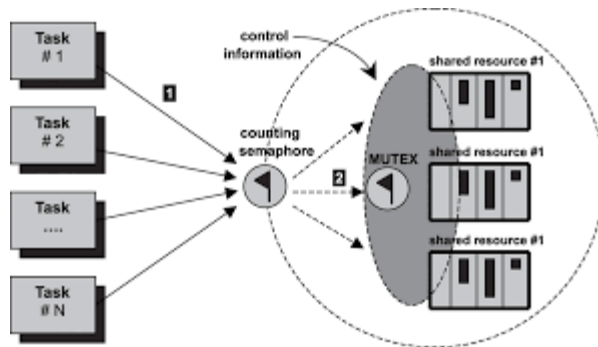
Todos estos servicios se manejan desde el kernel y se ofrecen al programador con librerías de alto nivel.



Comunicación y sincronización

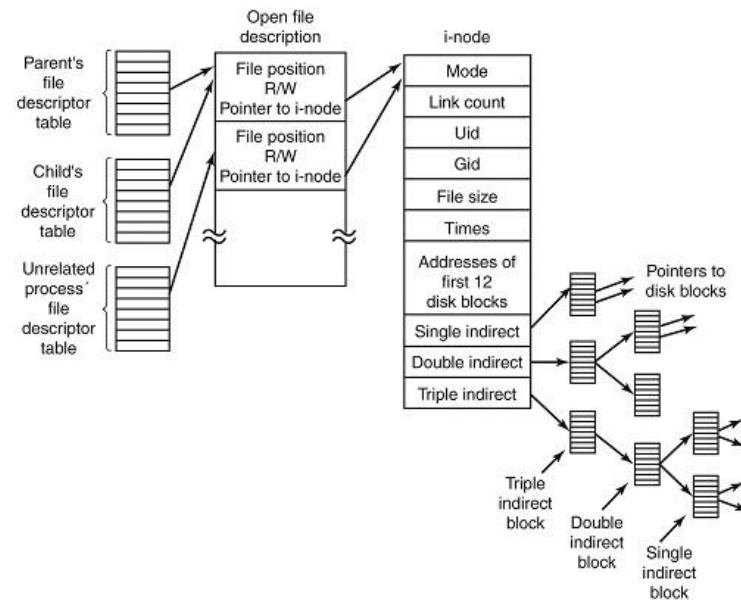
Cuando varios procesos o threads intentan acceder a un mismo recurso pueden surgir condiciones de carrera. Imaginemos dos procesos de compra web de billetes de avión y que no hubiese un mecanismo de bloqueo. Ambos podrían ofrecer el mismo asiento a dos usuarios entre el momento de la consulta y la finalización del proceso.

Estas situaciones son comunes en la programación concurrente y en el propio funcionamiento del Sistema Operativo. La única forma de evitarlas es proporcionando un mecanismo que asegure la ejecución de un fragmento de código sin que el sistema sea expulsado por otro que quiere acceso a los mismos recursos. El Sistema ofrece semáforos y mutexes como solución, su uso evita muchos problemas pero requiere una disciplina importante del programador.



Gestión de ficheros

Los sistemas necesitan un mecanismo de almacenamiento secundario no volátil para guardar datos o el código de los programas. En el pasado este papel lo desempeñaron las cintas y discos magnéticos. Sobre ellos, se construyó una abstracción, el fichero, que es un conjunto de información organizada accesible de forma secuencial y con unos metadatos asociados [nombre, fecha de creación, permisos]. Aunque la entidad fichero refleja su origen “magnético”, su flexibilidad es grande y ha permitido trasplantarla a tecnologías muy diferentes como los discos SSD, las memorias SD o en su día los CD-ROM. El Gestor de ficheros del Sistema Operativo oculta los detalles particulares de los dispositivos de almacenamiento y presenta la información a los usuarios con la abstracción fichero (FILE en C, por ejemplo). El estándar POSIX incluye todas las rutinas para su tratamiento.

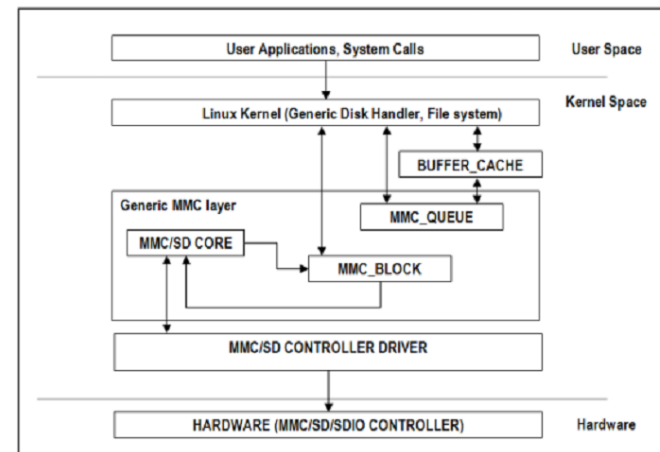


Más árboles...

Gestión de Entrada/Salida

La gestión de los dispositivos de Entrada/Salida es una de las labores más complejas por la gran variedad de fabricantes. Cada equipo ofrece una interfaz no estándar, que presentan unos registros en un circuito propio llamado controlador. La sintaxis y semántica de comunicación con ellos la abstrae un driver, un módulo especial que se linka dinámicamente con el kernel y permite poner en comunicación ese dispositivo específico. Gran parte de la complejidad de los kernel de Windows o Unix reside en la inmensa cantidad de dispositivos que son capaces de manejar.

El Sistema Operativo ofrece una abstracción de los dispositivos muy similar a la de los ficheros, de hecho la mayoría pueden tratarse como ficheros con algunas características especiales desde los programas de usuario.

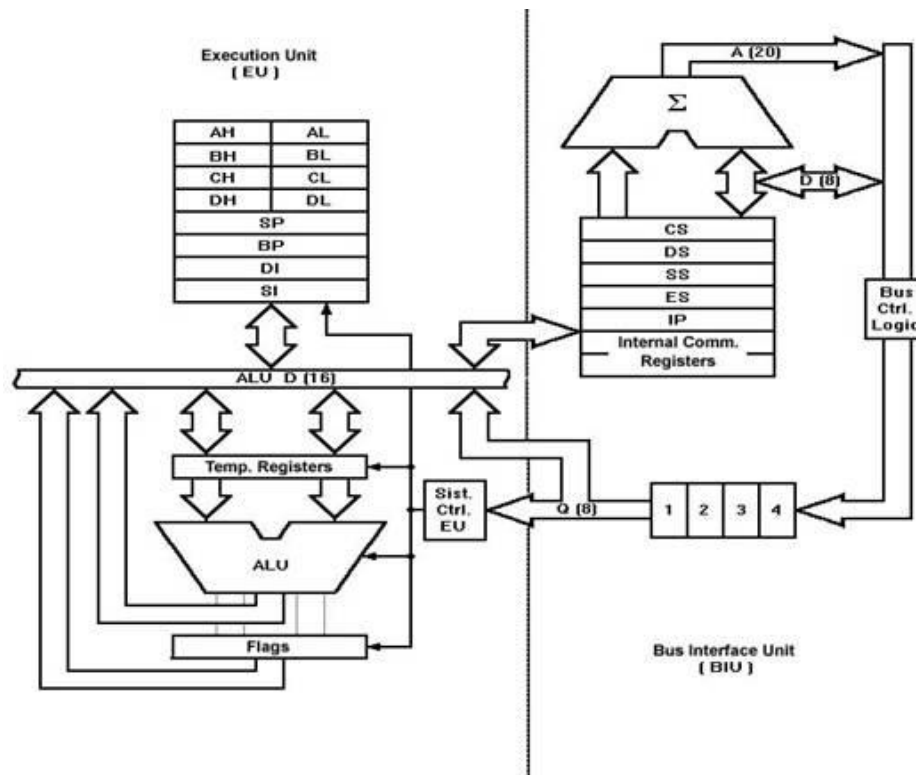


Fuente: [Texas Instruments](https://www.ti.com/wwww/wwww/en/products/embedded/index.tsp)

Arquitectura x86

Modelo del 8086. Arquitectura de 16 bits y bus de direcciones de 20 bits (podía manejar hasta 1MB de RAM)

- AX a DX: Registros de propósito general, accesibles en dos mitades de 8 bytes por compatibilidad con el 8080. AX es el acumulador.
- SP: Stack Pointer
- BP: Base Pointer (inicio del stack)
- DI, SI: Registros de indirección
- IP: Instruction Pointer, contador de programa.
- CS, DS, SS, ES: Registros de segmento.

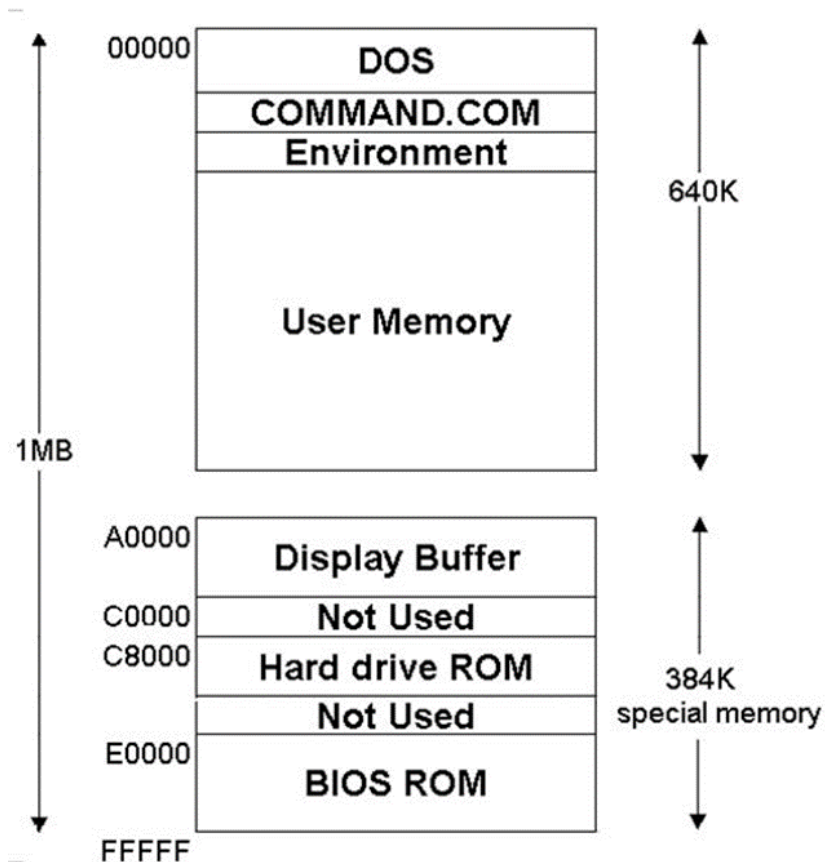


El 8086 no tenía modo supervisor (se introdujo en el 80286), de manera que el usuario tenía acceso a todos los recursos y direcciones

Arquitectura x86

Modelo de memoria del PC original con MS-DOS. El Sistema Operativo ocupaba las direcciones bajas y la memoria útil para el usuario llegaba solo hasta los 640 kB. Por encima se situaban la memoria gráfica, el driver de los discos y la BIOS en la parte superior del mapa.

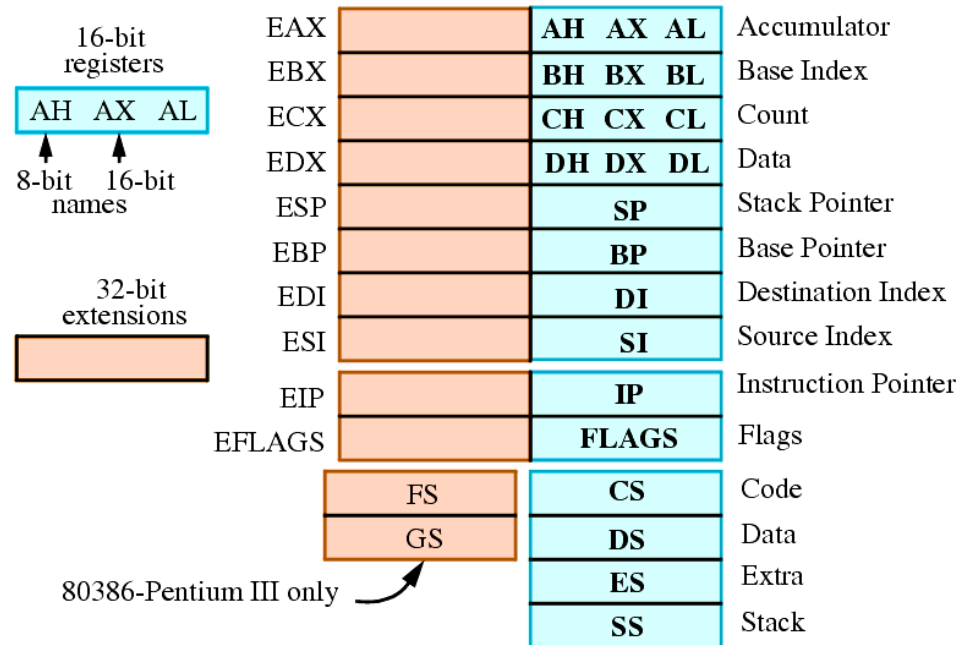
MS-DOS era monoproceso y no ofrecía protección por la inexistencia del modo supervisor en el 8086.



Arquitectura x86

Modelo del 80386. Arquitectura de 32 bits y bus de direcciones de 32 bits (podía manejar hasta 4GB de RAM)

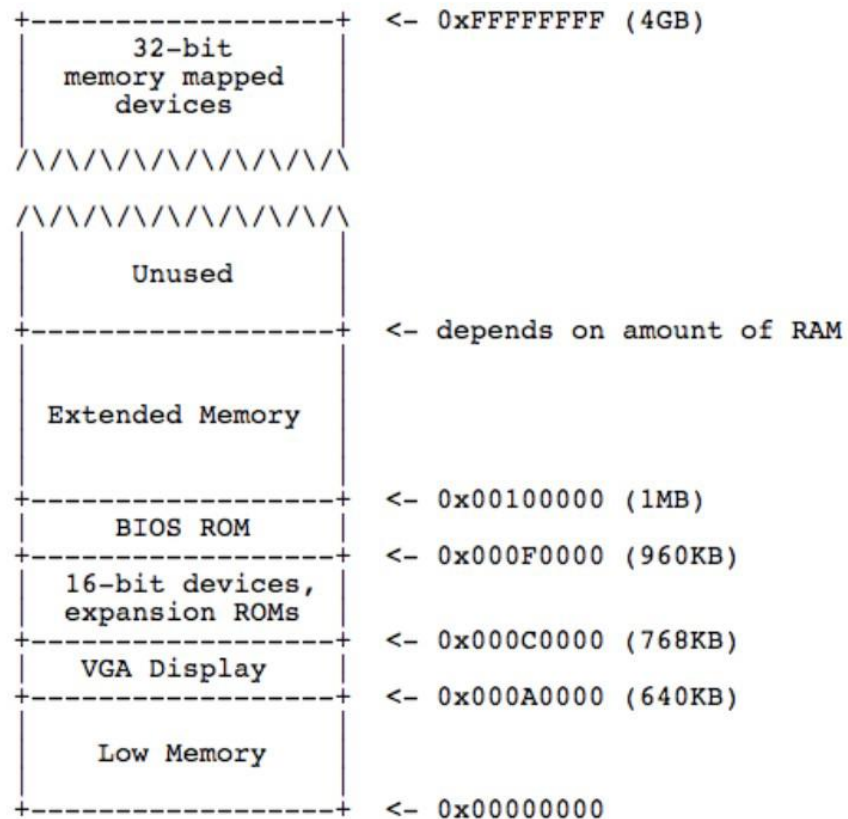
- EAX a EDX: Los registros se expanden a 32 bits, por compatibilidad hacia atrás se permite el acceso en modo 16 bits (AX, BX) y 8 bits en aquellos que lo soportaban en el 8086 (AH, AL, ... DH, DL)
- Los registros de manejo de direcciones también se amplían a 32 bits (ESP, EBP, EIP...) pero se permite usarlos con el nombre de 16 bits (SP, BP) para que puedan funcionar los ejecutables compilados con esa arquitectura.
- Hay tres registros especiales de control (CR0 a CR3). El modo supervisor se fija con un bit de CR0.



Arquitectura x86

En el modelo de 32 bits, la zona baja se distribuye de la misma forma que en los micros de 16 bits y los programas de usuario se cargan en la zona extendida, por encima de 1 MB.

Las direcciones más altas se reservan para el mapeo de dispositivos de entrada/salida.



Arquitectura x86

Modelo de arquitectura de 32 bits y bus de direcciones de 48 bits

- RAX a RDX: Registros expandidos a 64 bits, por compatibilidad hacia atrás se permite el acceso en modo 32, 16 y 8 en aquellos que lo soportaban.
- R8 a R15: Nuevos registros de propósito general de 64 bits.
- Los registros de manejo de direcciones también se amplían a 32 bits (ESP, EBP, EIP...) pero se permite usarlos con el nombre de 16 bits (SP, BP) para que puedan funcionar los ejecutables compilados con esa arquitectura.

General Purpose Registers

	<i>eax</i>	rax
	<i>ebx</i>	rbx
	<i>ecx</i>	rcx
	<i>edx</i>	rdx
	<i>esi</i>	rsi
	<i>edi</i>	rdi
	<i>ebp</i>	rbp
	<i>esp</i>	rsp
		r8
		r9
		r10
		r11
		r12
		r13
		r14
		r15

Floating Point Registers

<i>mm0/st0</i>
<i>mm1/st1</i>
<i>mm2/st2</i>
<i>mm3/st3</i>
<i>mm4/st4</i>
<i>mm5/st5</i>
<i>mm6/st6</i>
<i>mm7/st7</i>

63 0

Instruction Pointer

<i>eip</i>	rip
------------	-----

63 0

SSE Registers

<i>xmm0</i>
<i>xmm1</i>
<i>xmm2</i>
<i>xmm3</i>
<i>xmm4</i>
<i>xmm5</i>
<i>xmm6</i>
<i>xmm7</i>
<i>xmm8</i>
<i>xmm9</i>
<i>xmm10</i>
<i>xmm11</i>
<i>xmm12</i>
<i>xmm13</i>
<i>xmm14</i>
<i>xmm15</i>

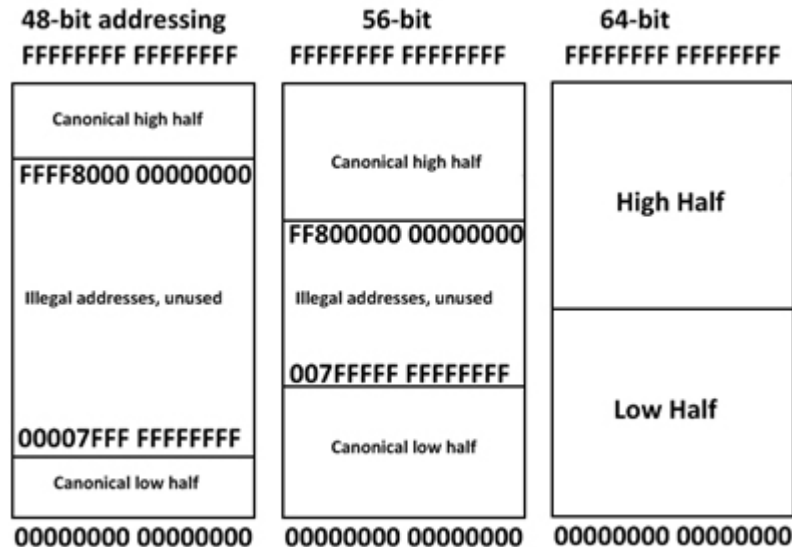
127 0

Registros para procesamiento de streaming

Arquitectura x86

Con 64 bits pueden direccionarse 16 EB (ExaBytes) pero los micros actuales no usan todos los bits del bus, sino 48. La arquitectura prevé las futuras ampliaciones como puede verse en la figura.

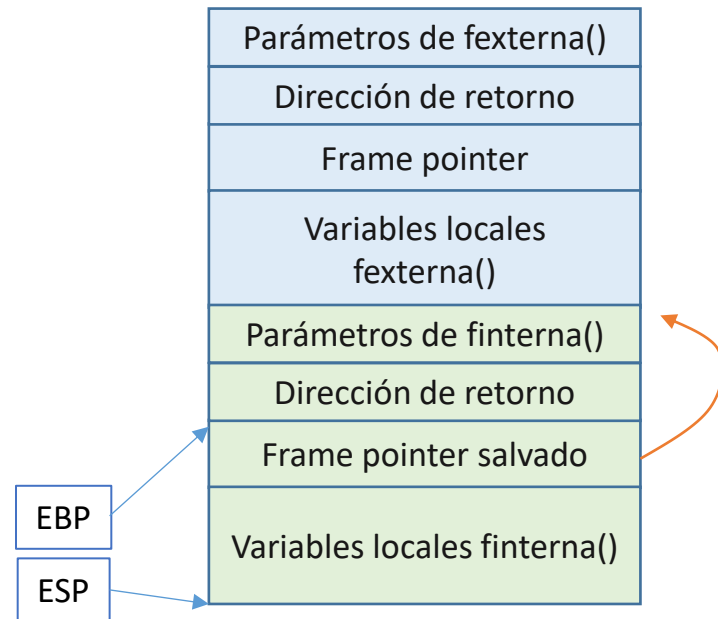
De cara a su uso por los sistemas operativos, el **espacio de direcciones se divide en dos mitades (low half y high half)**, la inferior se dedica a los procesos de usuario y la superior al kernel (se estudiarán los detalles en la lección 4).



Arquitectura x86

Manejo del stack

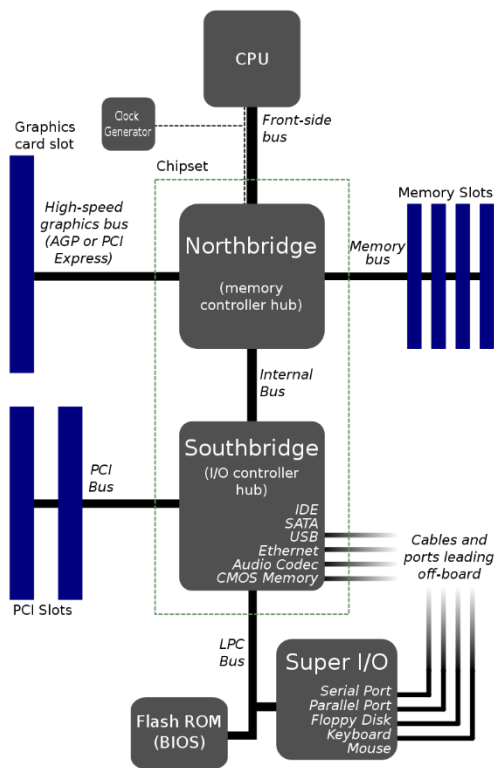
El manejo del stack en la arquitectura x86 es convencional. Hay un registro de stack SP (ESP o RSP según la arquitectura sea de 32 ó 64 bits) que apunta al último elemento añadido. Cuando se invoca una función, el stack almacena los parámetros de entrada, la dirección de retorno y las variables locales de la función. A este conjunto se le llama **frame** (marco). Para facilitar el manejo de la pila, los micros Intel incorporan un **frame pointer**, (registro %ebp) situado al inicio del marco de la función en el stack. El contenido del stack pointer varía durante la ejecución de la función, no así el del frame pointer. El frame pointer salvado contiene el valor del SP antes de la llamada a la función y se usa para restaurar este registro en el retorno.



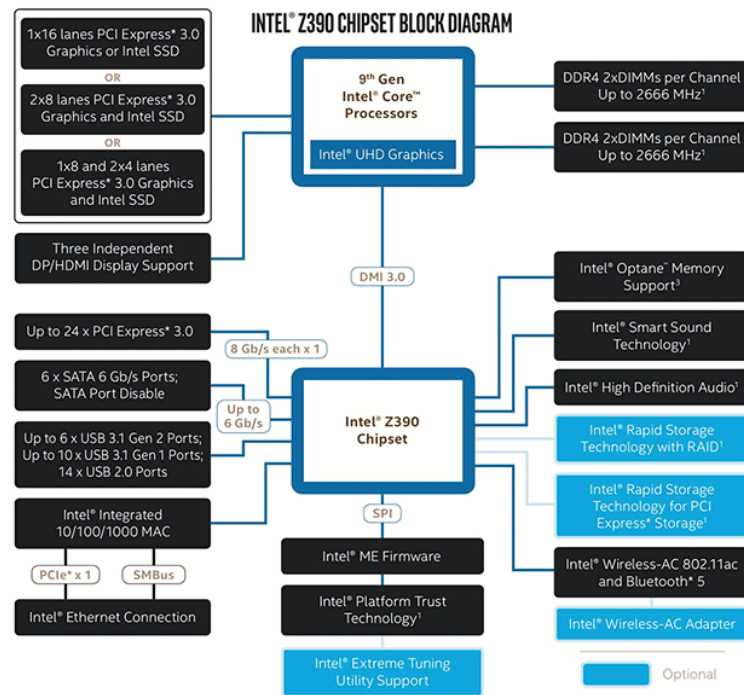
En este ejemplo la función finterna() ha sido invocada desde fexterna()

Ejemplo detallado completo en <https://www.cs.rutgers.edu/~pxk/419/notes/frames.html>

Arquitectura PC



Chipset hasta 2011 (primera generación i3, i5)



Chipset de la 9ª generación
Básicamente, el Northbridge se ha migrado al chip del microprocesador

Arquitectura ARM

ARM no es un fabricante de chips sino una arquitectura diseñada por una empresa británica del mismo nombre (Advanced RISC Machines), que la licencia a distintos fabricantes. Los microprocesadores Qualcomm Snap Dragon o Huawei Kirin tienen arquitectura ARM, también son ARM los micros de TSMC para el iPhone o los que instala Samsung en sus equipos. En Raspberry Pi se usa un único circuito fabricado por Broadcom que incluye un procesador ARM de gama baja junto con elementos (Ethernet, USB, GPU).

La historia de ARM es larga, actualmente va por la versión 8. Son micros de 64 bits, de arquitectura RISC. Los micros RISC tienen un juego de instrucciones de código máquina mucho más simple que el de los micros CISC como x86. Esto facilita el diseño hardware y los hace muy eficientes energéticamente. A cambio, los compiladores son más complejos.

El juego de instrucciones de ARM es del tipo Load and Store, no puede hacer operaciones directamente sobre contenidos de memoria, siempre tiene que llevarlos a un registro. Tampoco puede hacer operaciones entre dos direcciones de memoria, siempre deben pasar por un registro.

La [documentación del fabricante](#) es excelente. Puede bajarse el manual de arquitectura completo, son más de 8000 páginas.

Arquitectura ARM

General Registers and Program Counter Modes

Registros en modo usuario

User32	FIQ32	Supervisor32	Abort32	IRQ32	Undefined32
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)

Modos privilegiados

Program Status Registers

Flags

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

CPSR de usuario salvado en los modos privilegiados

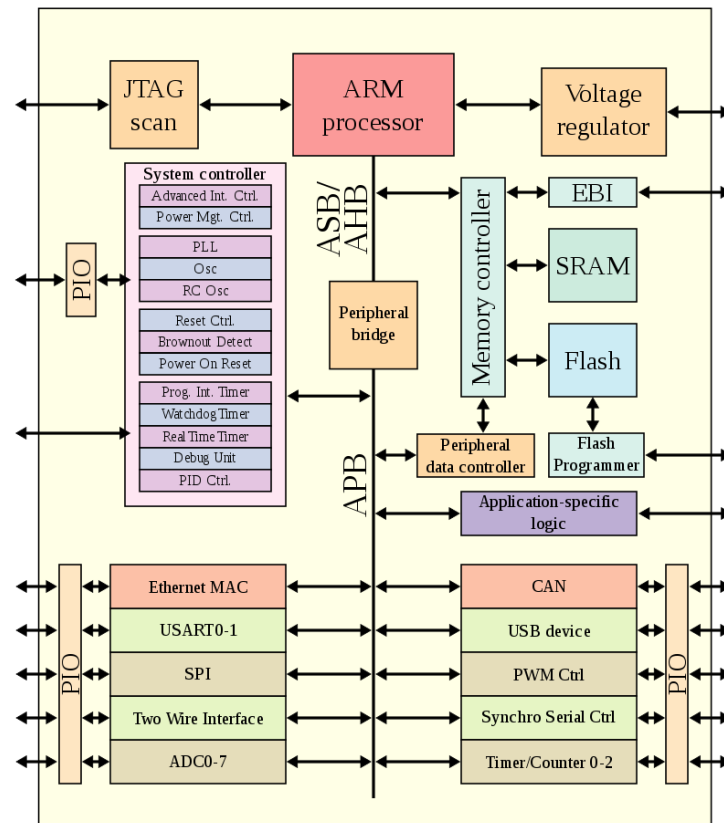
Arquitectura ARM

La arquitectura de un equipo basado en ARM refleja su evolución en el entorno de dispositivos móviles.

ARM permite manejar memoria virtual pero en los teléfonos móviles no usa swap como se hace en un PC de sobremesa, hay otros mecanismos.

Existen portados de Linux a arquitecturas ARM, como Raspbian para RaspBerry que usan swap de modo habitual.

En verano de 2020 Apple ha anunciado que migrará todos sus equipos de sobremesa a RAM



Arquitectura ARM



En 2011 Microsoft lanzó Windows RT que era un portado de Windows 8 a ARM. Fue un fracaso comercial pero no porque funcionase mal. La razón es que los códigos máquina de ARM e Intel no son compatibles, solo podían instalarse aquellas aplicaciones migradas a ARM por Microsoft o terceros. En contraste, Windows 10 puede ejecutar aplicaciones legacy Win32 sin ningún problema.

Esta historia ilustra un hecho muy importante. El éxito de un Sistema Operativo no reside tanto en su bondad técnica como en el ecosistema de desarrolladores dispuestos a trabajar en él. Este mismo motivo llevó al cierre a Sistemas Operativos móviles como Firefox OS o Windows Mobile.

Intel vs ARM

```
int i,j;
void main()
{
    i = 0;
    for (j=0;j<10;j++)
        i++;
}
```

x86

```
0000000000001129 <main>:
1129: f3 0f 1e fa      endbr64
112d: 55              push    %rbp
112e: 48 89 e5         mov     %rsp,%rbp
1131: c7 05 dd 2e 00 00 movl    $0x0,0x2edd(%rip) # 4018 <i>
1138: 00 00 00         jmp     1165 <main+0x3c>
113b: c7 05 cf 2e 00 00 movl    $0x0,0x2ecf(%rip) # 4014 <j>
1142: 00 00 00
1145: eb 1e           jmp     1165 <main+0x3c>
1147: 8b 05 cb 2e 00 00 mov     %eax,0x2ecb(%rip),%eax # 4018 <i>
114d: 83 c0 01         add     $0x1,%eax
1150: 89 05 c2 2e 00 00 mov     %eax,0x2ec2(%rip),%eax # 4018 <i>
1156: 8b 05 b8 2e 00 00 mov     %eax,0x2eb8(%rip),%eax # 4014 <j>
115c: 83 c0 01         add     $0x1,%eax
115f: 89 05 af 2e 00 00 mov     %eax,0x2eaf(%rip),%eax # 4014 <j>
1165: 8b 05 a9 2e 00 00 mov     %eax,0x2ea9(%rip),%eax # 4014 <j>
116b: 83 f8 09         cmp     $0x9,%eax
116e: 7e d7           jle     1147 <main+0x1e>
1170: 90              nop
1171: 90              nop
1172: 5d              pop     %rbp
1173: c3              retq
1174: 66 2e 0f 1f 84 00 00 nopw    %cs:0x0(%rax,%rax,1)
117b: 00 00 00
117e: 66 90           xchgb   %ax,%ax
```

ARM

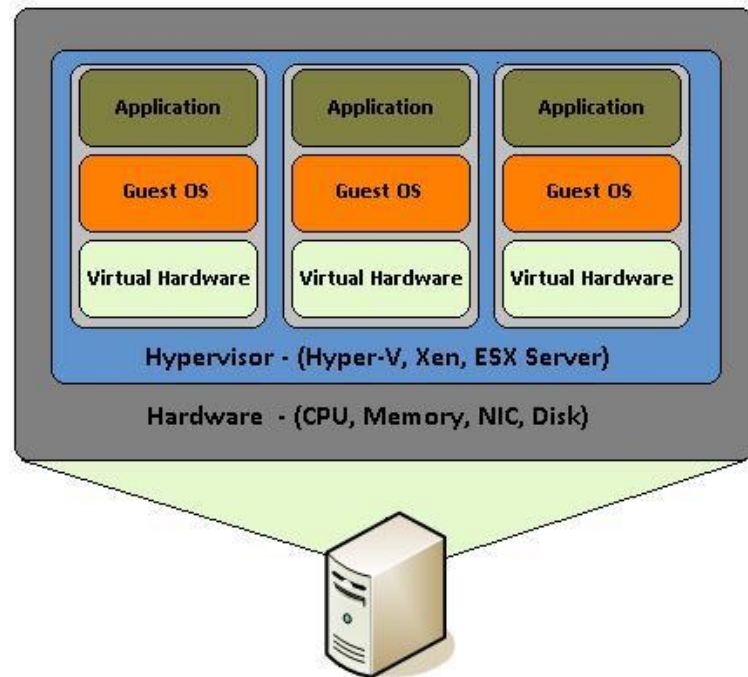
```
000103d0 <main>:
103d0: e52db004      push    {fp}           ; (str fp, [sp, #-4]!)
103d4: e28db000      add     fp, sp, #0
103d8: e59f305c      ldr     r3, [pc, #92]   ; 1043c <main+0x6c>
103dc: e3a02000      mov     r2, #0
103e0: e5832000      str     r2, [r3]
103e4: e59f3054      ldr     r3, [pc, #84]   ; 10440 <main+0x70>
103e8: e3a02000      mov     r2, #0
103ec: e5832000      str     r2, [r3]
103f0: ea000009      b       1041c <main+0x4c>
103f4: e59f3040      ldr     r3, [pc, #64]   ; 1043c <main+0x6c>
103f8: e5933000      ldr     r3, [r3]
103fc: e2833001      add     r3, r3, #1
10400: e59f2034      ldr     r2, [pc, #52]   ; 1043c <main+0x6c>
10404: e5823000      str     r3, [r2]
10408: e59f3030      ldr     r3, [pc, #48]   ; 10440 <main+0x70>
1040c: e5933000      ldr     r3, [r3]
10410: e2833001      add     r3, r3, #1
10414: e59f2024      ldr     r2, [pc, #36]   ; 10440 <main+0x70>
10418: e5823000      str     r3, [r2]
1041c: e59f301c      ldr     r3, [pc, #28]   ; 10440 <main+0x70>
10420: e5933000      ldr     r3, [r3]
10424: e3530009      cmp     r3, #9
10428: daffffff      ble     103f4 <main+0x24>
1042c: e1a00000      nop
10430: e28bd000      add     sp, fp, #0
10434: e49db004      pop     {fp}           ; (ldr fp, [sp], #4)
10438: e12fff1e      bx      lr
1043c: 0002102c      .word   0x0002102c
10440: 00021028      .word   0x00021028
```

Virtualización

Para termina la lección, una nota sobre algo que no veremos en este curso pero de lo que sí haremos uso es la virtualización de HW (con OracleVM y QEMU).

Los microprocesadores modernos ofrecen la posibilidad de usar un hipervisor, una combinación de SW y HW que presenta recursos de HW virtual a los sistemas operativos invitados que después se instalan sobre esta capa.

En los equipos Intel, el hipervisor se ejecuta en el anillo 1 de privilegio.



La terminología es engañosa. La gestión de memoria virtual no tiene nada que ver con los recursos de virtualización de la CPU.

Virtualización

El hipervisor o monitor es un software que permite virtualizar el hardware y poder instalar en una misma máquina física distintos sistemas operativos. Los más antiguos son los de tipo I, que se ejecutan fuera del SO, directamente sobre el hardware (por eso se conocen como bare metal). Se emplean sobre todo en los centros de datos.

Los de tipo II se ejecutan sobre el sistema operativo anfitrión (host) sobre el que se pueden instalar los invitados. A esta categoría pertenece Oracle VM y QEMU.



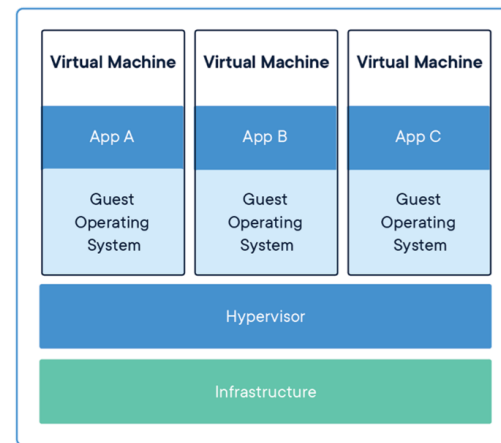
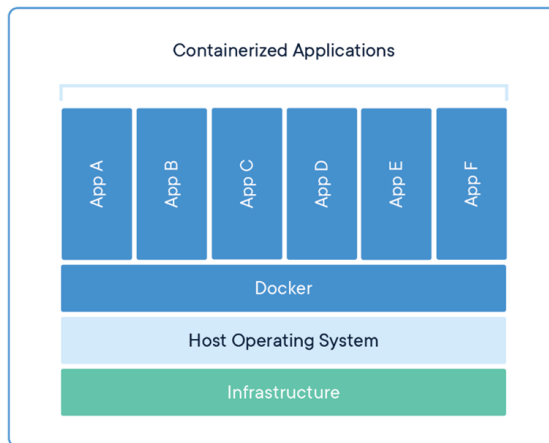
Tipo I



Tipo II

Virtualización

Un contenedor es un paquete de software que contiene todo el entorno necesario para su ejecución (ejecutables, librerías, permisos, variables, ...). Se ejecutan en un servidor (como *Docker Engine*) que se monta sobre el Sistema Operativo anfitrión. Los contenedores comparten el SO anfitrión, pero Docker consigue que los procesos de cada contenedor queden aislados de los otros usando la funcionalidad namespaces de Linux y controla el uso máximo de recursos con cgroup.



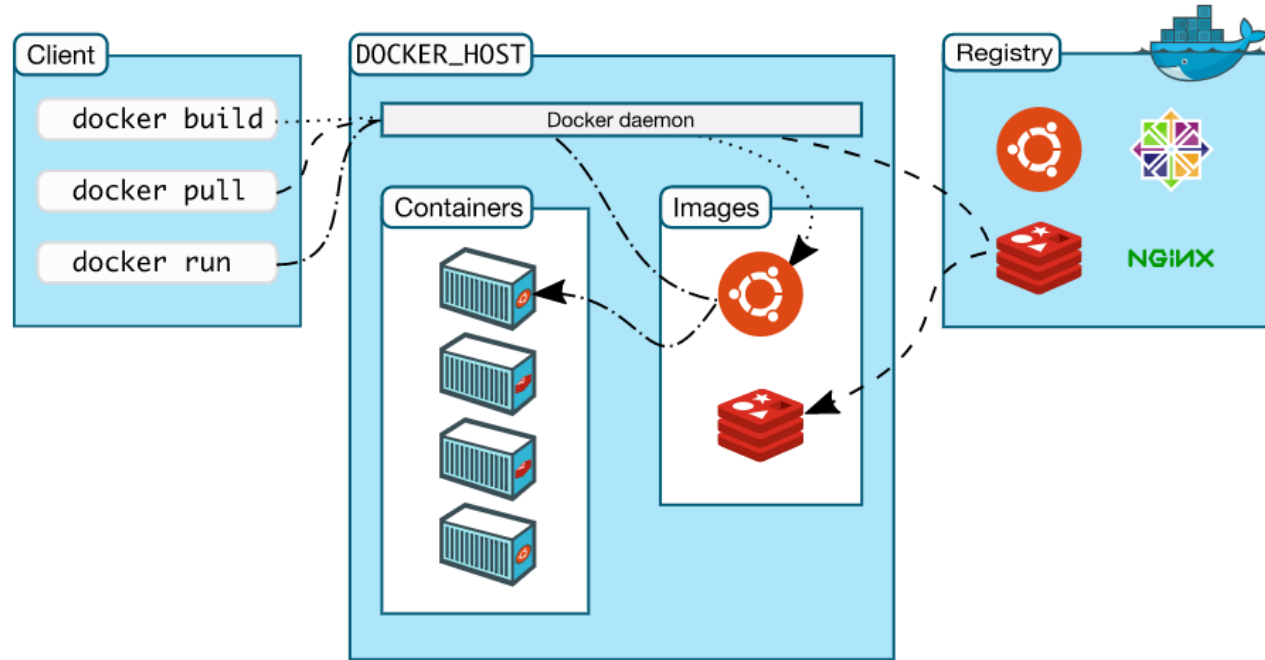
Virtualización

Docker y otros productos similares se apoyan en la virtualización a nivel del sistema operativo que ofrece Linux. Mientras un proceso puede acceder a todos los recursos de la máquina (siempre que esté autorizado) en un contenedor solo puede ver los que se conceden de antemano al contenedor, por eso da la sensación de que hay varias máquinas en paralelo, pero solo se está ejecutando una instancia del sistema operativo.

Una limitación importante es que no podemos ejecutar contenedores de un sistema operativo diferente al del host (sí pueden diferir las versiones). Si tenemos una máquina Linux, los contenedores no pueden lanzar aplicaciones Windows y viceversa. Esto no ocurre con las máquinas virtuales.

Con la tecnología de contenedores, por el contrario, podemos lanzar muchas más instancias de un proceso que las posibles con una máquina virtual en una misma máquina física. En cada VM tenemos que reservar CPU, RAM y disco, que se asignan aunque no estén en uso. En Docker, es el motor el que optimiza ese uso. El arranque es también rapidísimo, aunque los contenedores funcionan sin estado, es decir, en el siguiente arranque no tienen persistencia.

Virtualización



Docker funciona con un modelo cliente-servidor. El cliente puede ser una Shell de comandos o una API y no tiene por qué estar en la misma máquina que el demonio. Registry es el repositorio de imágenes.