

## Unidad 1: ARQUITECTURA DE LAS COMPUTADORAS PERSONALES

1.1 – Características, Arquitectura básica de procesador Intel Pentium y descripción funcional: Decodificador de Instrucciones, Unidad de Control, ALU, etc. Modo de trabajo real: Modelo de programación, Direcciones de memoria, Interrupciones y Excepciones. Pipeline, Cache y TLB.

1.2 – Modo protegido: Introducción, registros habilitados y traslación de direcciones lógicas a físicas. Segmentación: concepto, descriptores de segmento y manejo de memoria. Tipos de descriptores de segmentos, tablas de descriptores globales y locales. Paginación. Concepto de página, tablas y registros de soporte. La operación de paginación, descriptores de directorio de páginas y tabla de páginas, estructura y acceso. Uso del TLB.

1.3 - Sistema de protección: por segmentación, niveles de privilegio y restricción de acceso a los segmentos. Cambios de nivel de privilegio, puertas de llamada. Protección por paginación. **Manejo de interrupciones en modo real y protegido: interrupciones y excepciones. Concepto de excepción. Aplicación de las excepciones.**

1.4 – Multitarea: Introducción. **Métodos de planificación para sistemas operativos multiusuario.** Registros de soporte y descriptores relacionados a multitarea. Cambio de tarea con y sin puerta de tarea

Unidad	Tema	Libros, capítulos y secciones
1.1	Arquitectura CPU	Godse, A.P. <i>Microprocessors and Microcontrollers Systems, 3rd Ed. revised.</i> Technical Publications. 2008. <b>Secciones 1.2, 1.3 y 1.5 al 1.8.</b>
1.2	Memoria Virtual	Godse, A.P. <i>Microprocessors and Microcontrollers Systems, 3rd Ed. revised.</i> Technical Publications. 2008. <b>Secciones 4.1 al 4.8.</b>
1.3	Protección	Godse, A.P. <i>Microprocessors and Microcontrollers Systems, 3rd Ed. revised.</i> Technical Publications. 2008. <b>Secciones 4.9 al 4.14.</b>
1.4	Multitarea	Godse, A.P. <i>Microprocessors and Microcontrollers Systems, 3rd Ed. revised.</i> Technical Publications. 2008. <b>Secciones 5.1 al 5.4.</b>

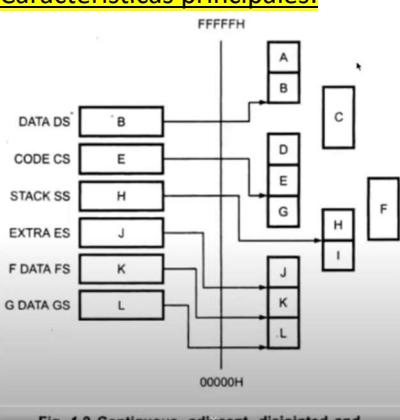
## -Clase del 17-03-2021 Modo Protegido.

Módulo de ejecución predictiva en función de saltos condicionales. Sin ver la parte de protección, el profe cuenta que había una vulnerabilidad que mediante instrucción de saltos a áreas protegidas porque eran de otro usuario, proceso o del Kernel, etc. ... Los datos se precargaban en cache, pero cuando se llegaba a decodificar y ejecutar esta instrucción daba una excepción y terminaba, pero lo malo era que luego se leía la cache y estaba disponible el contenido de la misma.

Cuando arranca la arquitectura IA-32, es casi como un 8088, una máquina con un direccionamiento de 1Mb, que no son continuos, sino en pedacitos de 64Kb. Si pongo el 1er bit de CR0 a 1, paso a modo protegido, donde tengo todas las funcionalidades y puedo direccionar 4 Gb de RAM, y tengo visibilidad de algunos registros Ej: GDTR, IDTR, RTDL, TR (Task Register: se utiliza para realizar cambios entre tareas)

El 1er bit de CR0 lo pone en alto, una aplicación que tiene permitido acceder a ese registro. Cuando arranca el SO, arranca en modo real y cambia a protegido, para aprovechar las bondades de todo el hardware.

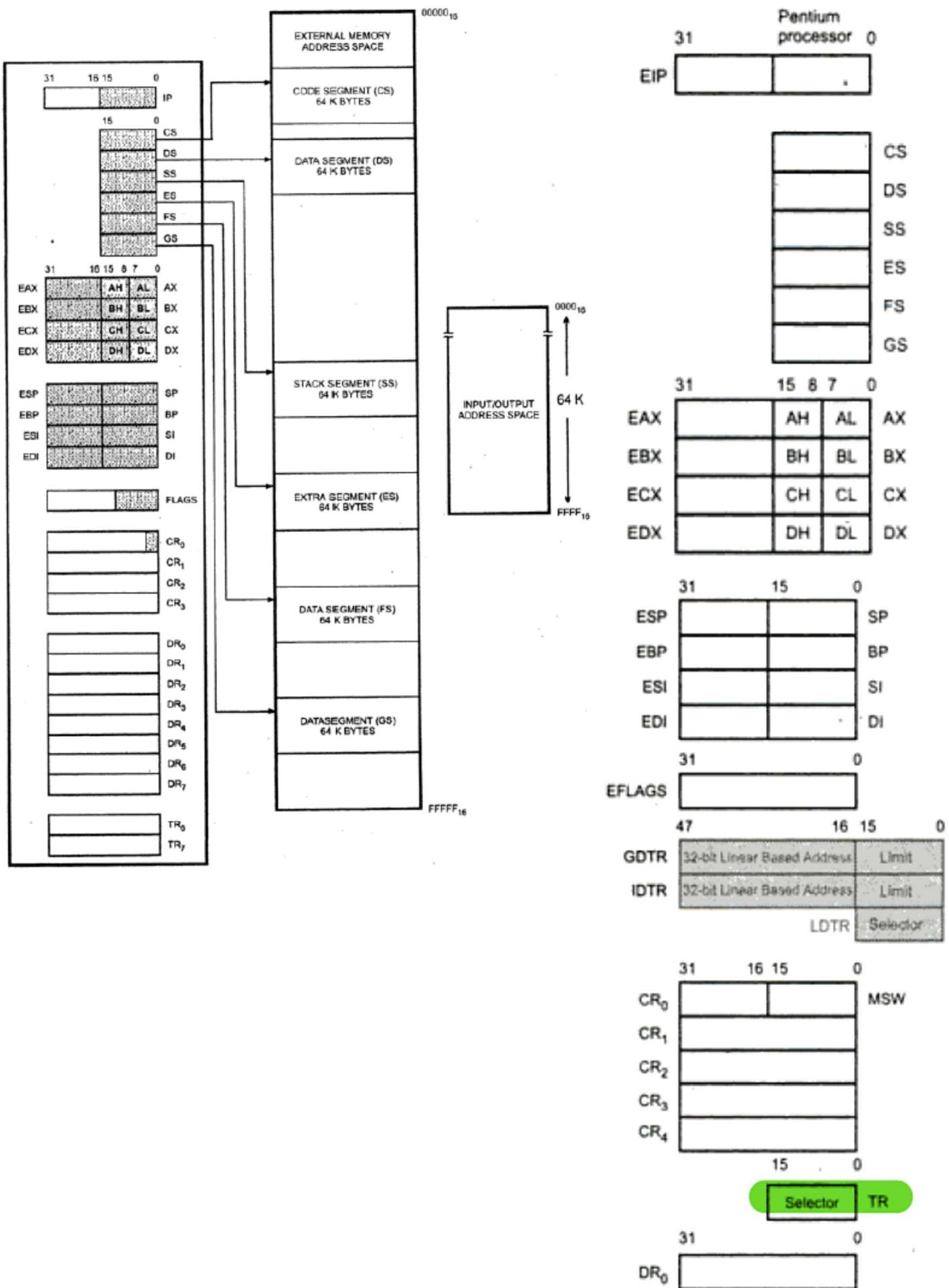
### Características principales:



Aparte de direccionar 4Gb. El M. Protegido también incorpora la aislación/protección entre tareas. Significa que, si tengo más de un programa corriendo, es imposible que uno escriba o lea información del otro. En modo Real, yo podría de acuerdo a los valores que se colocuen en los registros de segmento, se podrían solapar, y podría escribir en un área de datos que era de otro programa. En modo Protegido, se agrega una funcionalidad, que directamente por hardware si yo intento acceder a una posición de memoria que no me pertenece da una excepción de hardware y termina la aplicación, no puede acceder.

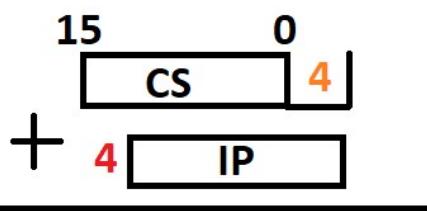
Otra característica interesante, es que tiene 4 modos de funcionamiento, o niveles de privilegio. Un nivel tiene acceso a todo el Set de instrucciones, sobre todo se refiere a instrucciones de entrada-salida. Mientras que el último nivel no tiene acceso a eso, tiene un set de instrucciones recortado, para evitar que haga lo y rompa alguna aplicación. Lo que termina pasando normalmente, es que

los SO trabajan en modo privilegiado o modo Kernel, donde tienen acceso a todo el set de instrucciones. Mientras que mi proceso, yo usuario, cuando escribo un programa termino trabajando en el modo menos privilegiado y no puedo hacer líos por más que quiera. Además, que tengo inhabilitado escribir o leer posiciones de memoria que son de otro programa, tampoco puedo hacer entradas-salida o instrucciones que modifiquen por ejemplo los 6 registros de segmentos (CS, DS, SS, ES, FS, GS). En modo usuario no tengo posibilidad de modificarlos.



Estas son las ventajas de trabajar en modo protegido. Hay 4 niveles, con que hubieran 2 nos arreglamos, pero Intel tiene 4. Los intermedios se pueden usar para drivers, por ejemplo, pero depende del SO que usemos.

## Direccionamiento.



**20 bits**

siendo de 16 bits, pero la diferencia es que, ahora el desplazamiento, ósea el offset es de 32.

Por ejemplo, el SP stack pointer, ahora se llama ESP Extended Stack Pointer, y en vez de tener 16 tiene 32 bits. Lo mismo que para el IP y el resto de los registros. OBSERVAR GRAFICA ANTERIOR. Solo los R. de segmento siguen con 16, los otros no. Y se pueden ver algunos registros más. (GDTR, IDTR, RTDL, TR).

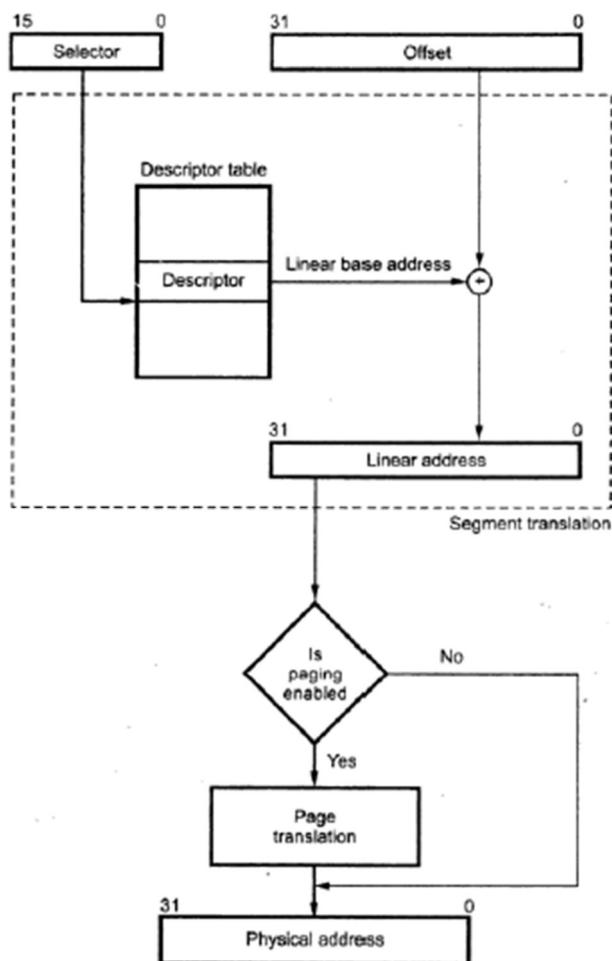


Fig. 4.2 Address translation overview

Antes era fácil, yo tenía el CS de 16 Bits, esto lo desplazaba 4 lugares y después le sumaba el Offset, que en este caso era IP (instruction Pointer - el Contador del programa). Esto era en modo real, y nos daba la dirección física. El resultado da 20 bits, puede direccionar 1Mb, pero como fijaba el CS, puedo direccionar solo 64Kb. Porque de los 20 bits, los 4 más significativos no cambian en la suma. Queda  $2^{16}$ .

En M. Protegido yo en realidad puedo acceder a 4Gb, los registros de segmento siguen

siendo de 16 bits, pero la diferencia es que, ahora el desplazamiento, ósea el offset es de 32.

Ahora en vez de tomar el segmento y desplazarlo 4 bits y sumar el offset, yo ahora tengo un segmento que sigue teniendo 16 bits, pero un offset de 32.

Acá hay una salvedad, que no es que haya un registro que se llame "SELECTOR", ni un registro que se llame "OFFSET", acá sigue siendo como en M. Real, si yo busco por ejemplo una instrucción, el registro que va en el selector es el CODE Segment CS, y el registro que va en el offset es el IP. U otro ejemplo, si estoy haciendo un push o pop de la pila, en el selector va el Stack Segment y en el offset el Stack Pointer. (ESP para este caso de M. Protegido). Entonces los nombres Selector y Offset están puestos de manera genérica, no un registro con ese nombre.

Acá viene la magia. El selector en vez de desplazarse, lo que hace es, con el valor que tiene cargado, busca dentro de una tabla (que se llama tabla de descriptores), la cual esta ubicada en la memoria Principal (la RAM). Es una tabla donde tengo información, que esta grupada de a varios Bytes y se llaman descriptores. Que básicamente describen (mapea) porciones/partes o segmentos de memoria. En RAM tengo una tabla con "n" elementos (grupos de bytes) llamados descriptores.

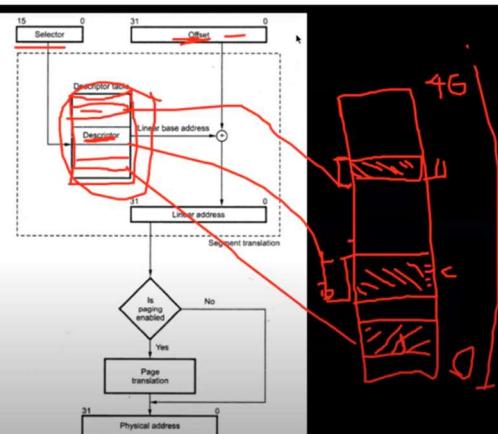
Dentro de cada uno de estos descriptores hay una dirección base, un límite y unos bits más que tienen que ver con los niveles de privilegio. Por cada segmento de la memoria principal voy a tener un descriptor. Si al sumar la base + Offset excede el límite se produce una excepción por hardware.

Acá no tenemos un tamaño fijo del segmento, antes eran de 64Kb en modo real. acá pueden tener los 4Gb un segmento o 4Kb lo que yo quiera.

¿Por qué? Porque esto en teoría el SO, o la persona que lo va a pasar a modo protegido, antes debería en la memoria principal (la RAM) armar esta estructura, esta Tabla. Hay un proceso para cambiar de M. Real a Protegido, pero primero lo que se hace es armar la tabla de descriptores. Entonces, al cambiar a M. Protegido los selectores apuntan a un valor coherente.

El offset del segmento de datos, depende de qué tipo de direccionamiento quiero, igual que en modo real. Relativo, absoluto, si la posición va en el argumento de la operación Read o Write, ese offset dependería un poco de algunas otras cosas, pero en definitiva cae dentro del área de datos.

Esto es una especie de direccionamiento indirecto, a través de una tabla que previamente se carga en RAM, de alguna manera se tiene mapeado todo el espacio de direcciones de 4Gb. Donde tengo áreas de Código, Datos, Pila, etc.



Cuando el SO arma esta tabla, lo hace de tal manera que los segmentos están separados y no se solapan, antes de cambiar a modo Protegido.

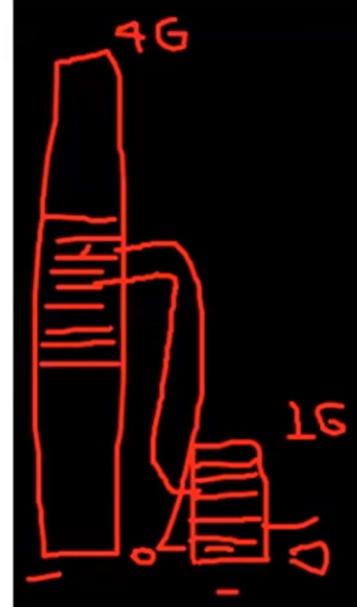
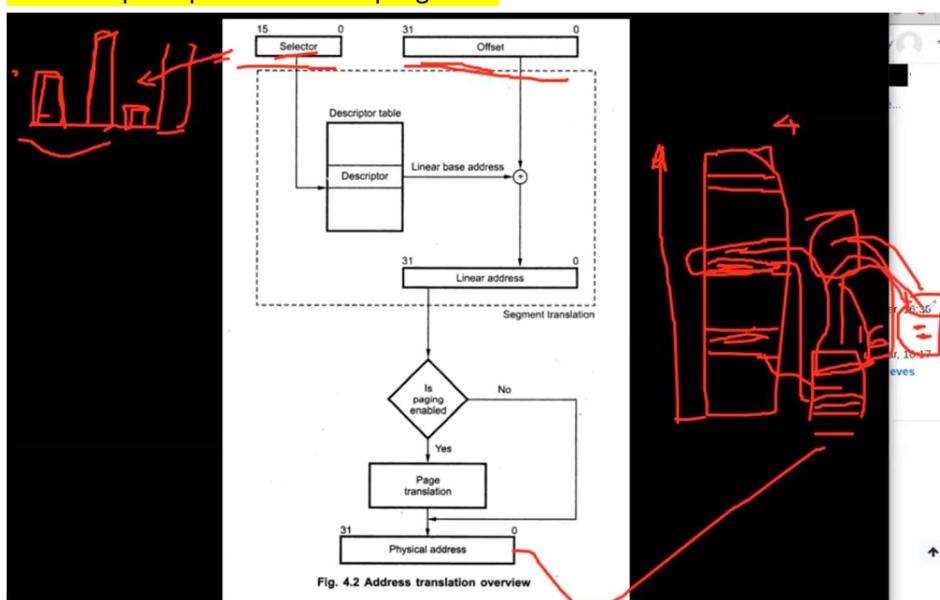
Todo esto genera una dirección de 32 bits, llamada Direc. Lineal. Arriba, donde está el procesador, partimos de la dirección VIRTUAL o Lógica. Desde el punto de vista del procesador, él se cree que tiene, por ejemplo, un área de memoria única para su programa, otro espacio de direcciones distinto para su pila, otro para los datos, etc. pero en definitiva van a parar todos a una única memoria unidimensional.

Arriba parte de lo que el procesador cree que ve, una dirección "virtual" ficticia, al medio lo acomodo todo en una sola dirección lineal de 0 a 4Gb

Esto es obligatorio en INTEL, en modo protegido, hago si o si Traslación de direcciones usando Segmentación, que se llama a esta técnica de traslación. **Trasladó que el procesador ve "n" espacios distintos de memoria, uno para cada segmento. El procesador creería que tiene "n" buses distintos.**

Ósea que cuando está en Modo protegido, si o si, ocurre una traslación de direcciones llamada Segmentación. A partir de un direccionamiento lógico virtual que ve el procesador donde cree que tiene "n" espacios de direcciones, se acomodan todos los segmentos en una única memoria lineal con un bus de 32 bits. Esto terminaría acá, salvo que esté habilitada una 2da traslación llamada PAGINACIÓN. Si están algunos bits de control CR0 en alto, habilitados, tengo que realizar esta 2da traslación. Caso contrario, la dirección lineal equivale a la Física y los 32 bits van directamente al bus de direcciones, dirección FISICA.

**PAGINACION:** Se usa para lo siguiente. Ej: Si tengo un programa que me ocupa en memoria RAM mucho espacio, supongamos el office, (Tener en cuenta que para ejecutar un programa tiene que estar cargado en la memoria principal, la RAM). Si tengo 1Gb de ram por ejemplo y cargo un programa grande, no me queda lugar para nada más. Pero que pasa si el programa, lo corto en pedacitos y a medida que quiero acceder a una parte del programa que no está, ahí si lo cargo en la memoria principal. Y si hay un pedazo del programa en la memoria principal y hace mucho tiempo no se usa, lo podría sacar. **Eso me permitiría no ocupar tanta memoria principal con un solo programa.**



Supongamos que tenemos la memoria Lineal de 4Gb que vemos, pero una memoria Real Física de 1Gb. La M. Física, para itel, la separo en pedacitos iguales de 4Kb y se llaman MARCO DE PAGINA, donde se van a alojar las páginas. Por otro lado, en la memoria Lineal, dentro de cada segmento, también divido en pedacitos de 4Kb, que se van a llamar PAGINAS. Entonces cada vez que quiera ejecutar una instrucción dentro de una pagina del segmento de código, la voy a elegir y guardar dentro de un MARCO de página libre. (o que hace mucho que no se utilice)

# SEGMENTACION

En el sistema cuando arranca, tengo una única tabla de descriptores globales. Se multiplica x8, porque cada descriptor tiene 8 Bytes. Ósea el índice “2”, en memoria esta 8 Bytes mas arriba que el índice “1”. Cada índice del selector se multiplica por el espacio que ocupa un descriptor para, partiendo del GDTR que da la base, obtener el offset de memoria necesario para acceder al descriptor deseado.

Cuando se obtiene la Dir. Lineal se hace un control de hardware, que el Offset no sobre pase el límite, si no ocurre una excepción de violación de segmento.

**Los CAMPOS adentro de cada tipo de DESCRIPTOR NO TIENE SENTIDO APRENDERLOS. Dijo el profe Taffe**

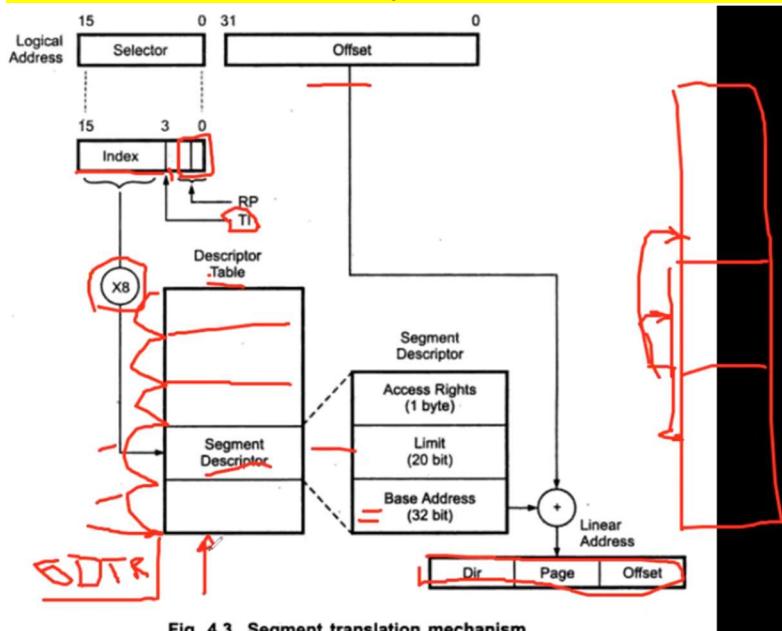


Fig. 4.3 Segment translation mechanism

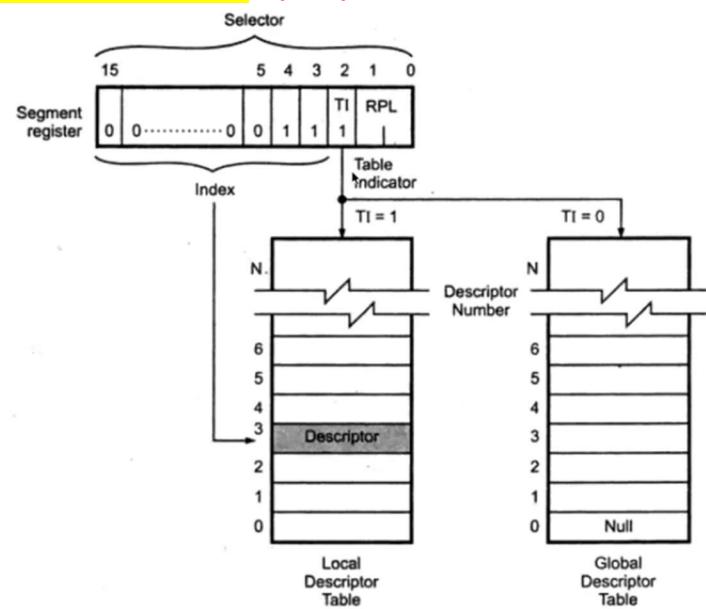
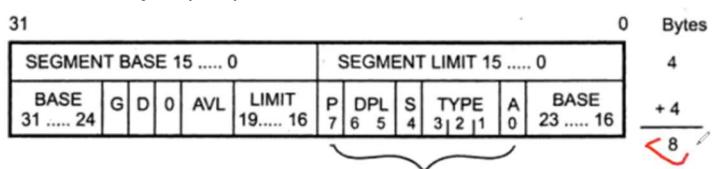


Fig. 4.4 Selector and descriptor tables

Hay una sola Tabla Global, hay una o varias Locales.

A modo de ejemplo presenta esta división.



BASE	Base Address of the segment
LIMIT	The length of the segment
P	Present Bit : 1 = Present 0 = Not present
DPL	Descriptor privilege Level 0 - 3
S	Segment Descriptor : 0 = System Descriptor 1 = Code or Data Segment Descriptor
TYPE	Type of segment
A	Accessed Bit
G	Granularity Bit : 1 = Segment length is page granular 0 = Segment length is byte granular
D	Default Operation Size (recognised in code segment descriptors only) 1 = 32-bit segment 0 = 16-bit segment
0	Bit must be zero (0) for compatibility with future processors
AVL	Available field for user or OS

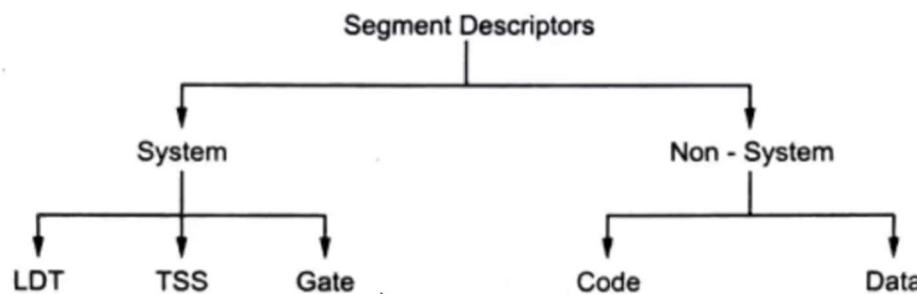
Presenta el formato genérico de un descriptor de segmento, hay un formato distinto para datos, para pila, para el SO.

**Lo importante son la base, el límite y los permisos.**

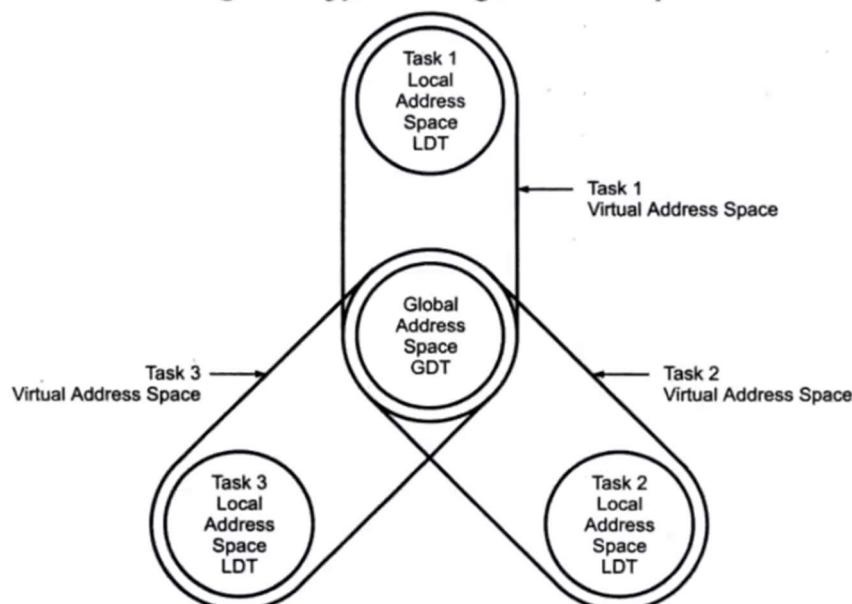
**Note :**  
In a maximum - size segment (i.e. a segment with G = 1 and segment limit 19 .... 0 = FFFFFH), the lowest 12 bits of the segment base should be zero. (i.e. segment base 11 .... 000 = 000H).

Fig. 4.5 General Segment Descriptor Format

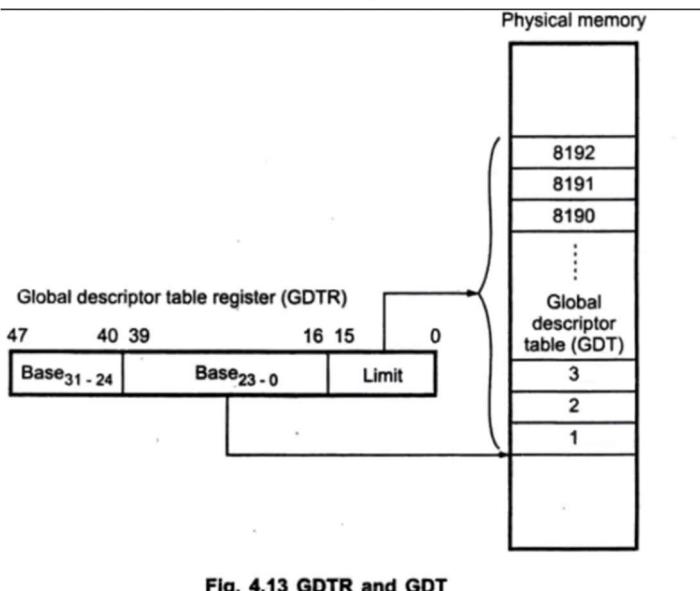
Hay distintos tipos de descriptores que tengo.



**Fig. 4.6 Types of segment descriptors**



**Fig. 4.12 Memory area shared by different tasks**



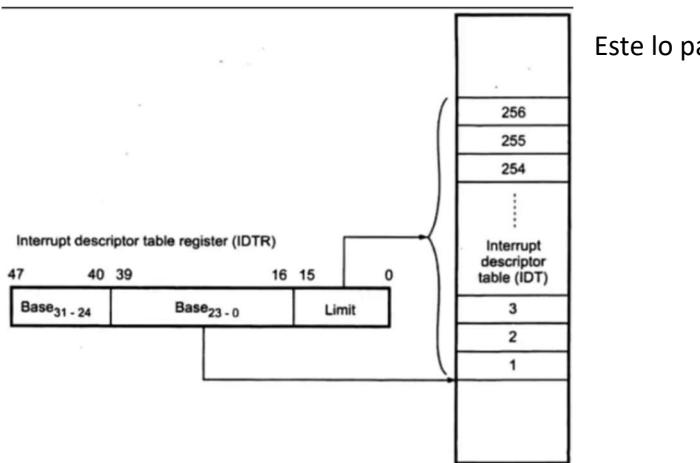
**Fig. 4.13 GDTR and GDT**

### TABLA de DESCRIPTORES GLOBALES

Cada Tarea/ Proceso / programa va a tener siempre acceso la TDG que es única para el sistema, pero cada tarea podría tener también una tabla de descriptores locales para “esa” tarea.

Tengo 16bits para el límite, ósea 8192 descriptores.

El único que tiene permiso para modificar este registro, es una rutina que corra con el mayor privilegio. ( Instrucción LGDTR “Load” – Un usuario común tiene ese set de instrucciones)



Este lo pasa de largo, ni lo menciona el Taffe

**Fig. 4.14 IDTR and IDT**

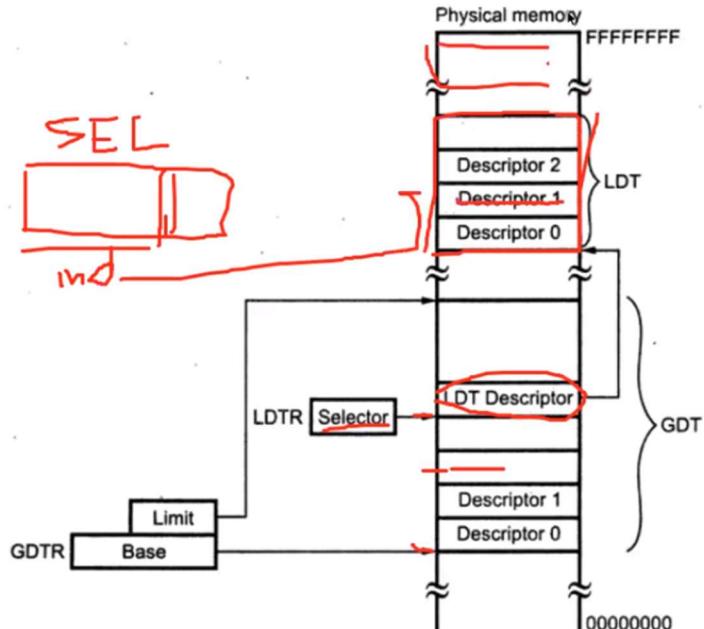


Fig. 4.15 Global and local descriptor tables

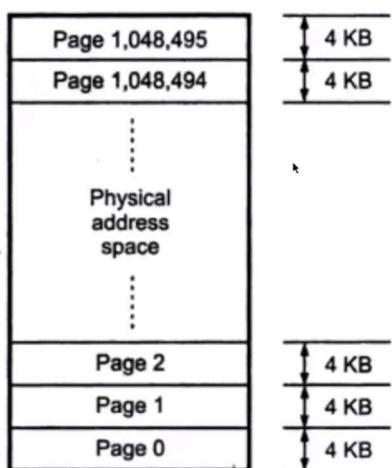


Fig. 4.18 Paged organization of the physical address space

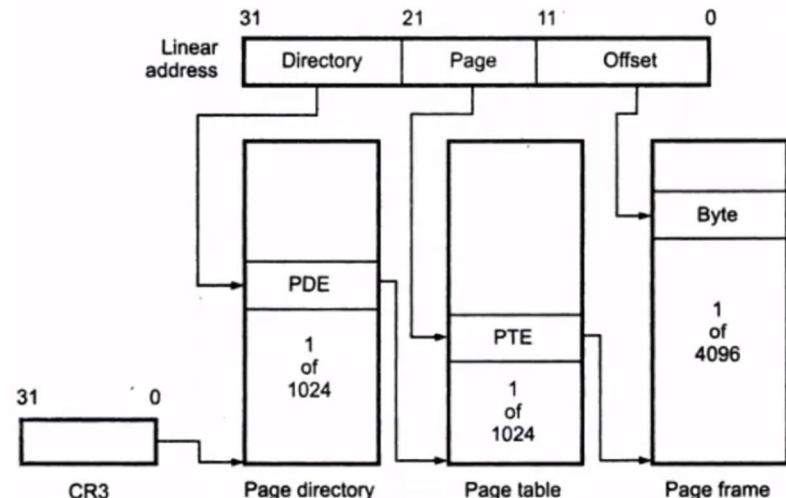


Fig. 4.20 Linear to physical address translation

Cuando  $TI = 1$  (índice de tabla está en alto), utilizo el selector cargado en LDTR primero en vez los 13 bits que actúan de índice en el selector del registro de Segmento. LDTR Es un registro físico de 16bits, que me apunta a un descriptor de un área de memoria, un segmento, pero no es de código ni datos, sino que hay información de la base y el límite de una tabla de descriptores locales.

Y en la LDT con el índice del R. de Segmento (1er selector) determino el descriptor buscado. Y este me mapea un segmento de la memoria principal.

Con esto puedo ampliar la tabla de descriptores, supongamos que no me alcanza con los 8192 descriptores de la GDT, le doy por ejemplo a cada uno de los 8192 procesos un LDT distinto a cada uno. Donde cada proceso tiene su tabla local independiente.

#### ESTO SERIA TODO EN CUANDO A SEGMENTACION

Todo esto es memoria lineal = física, si no tengo paginación. Si la paginación esta habilitada, a este direccionamiento lineal, le voy a hacer una 2da translación de memoria.

La paginación esta organizada en bloques de 4Kb, la memoria virtual tiene páginas y la memoria física donde se van alojar esas páginas, tiene marcos de página.

Los primeros 10 bits los utilizo de offset, para una tabla llamada directorio de páginas que también está en la Memoria principal, en la RAM. Y con esto tengo 1 de 1024 posibles entradas.

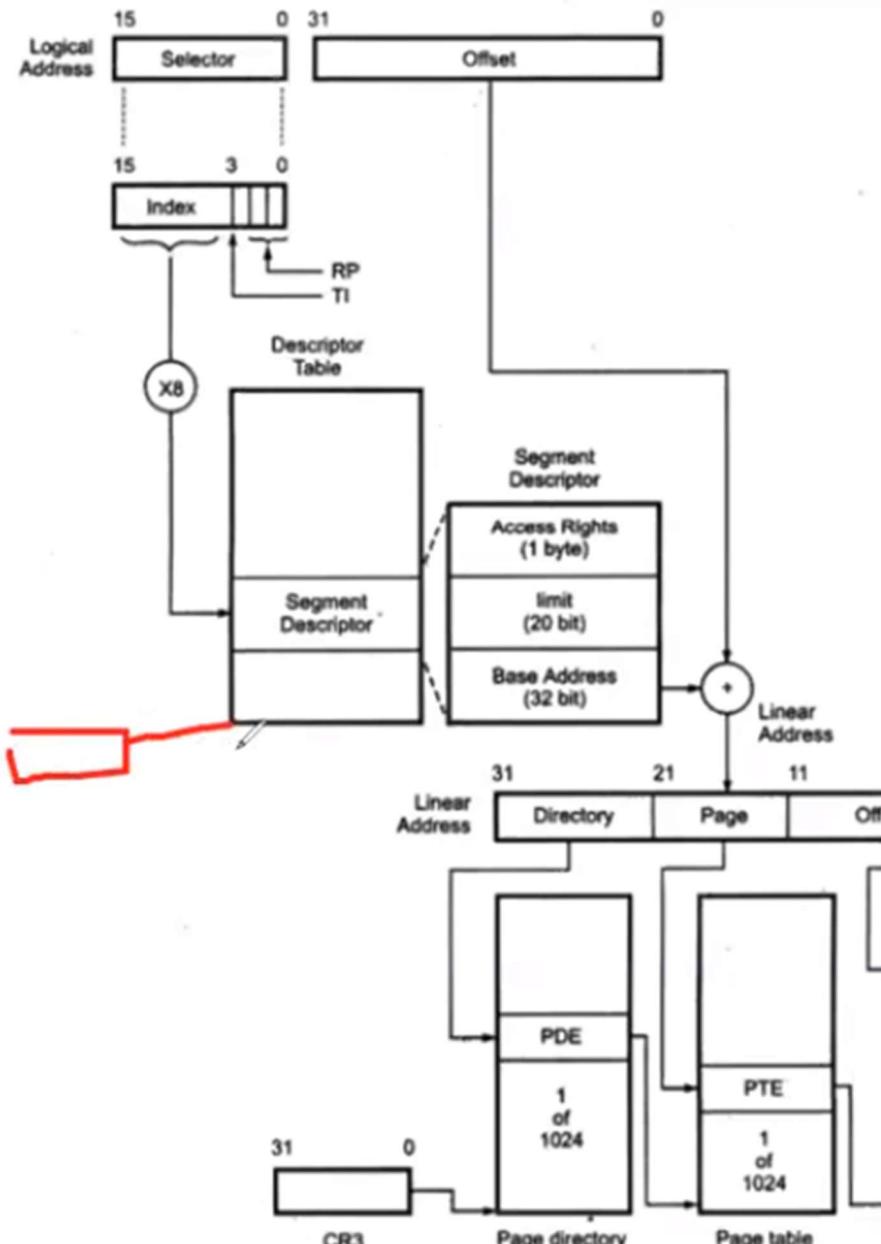


Fig. 4.24 Protected mode address translation

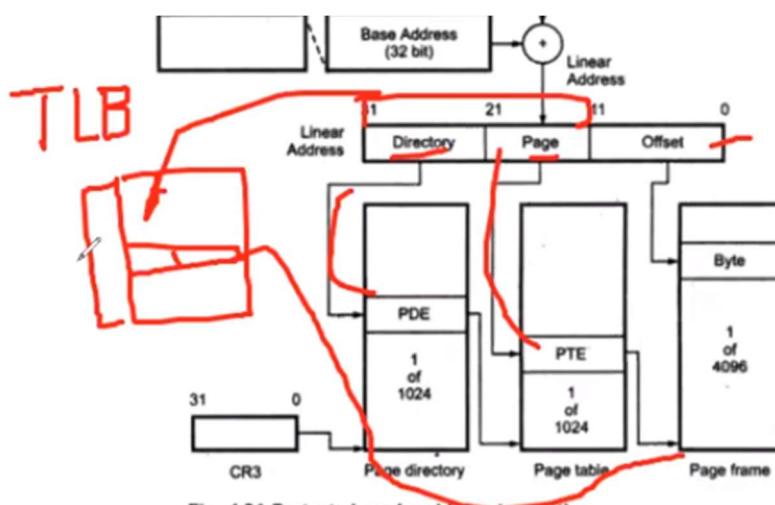


Fig. 4.24 Protected mode address translation

Falta explicar en esta imagen el comportamiento de la tabla global y local de descriptores.

¿Cuántos accesos a memoria tengo que realizar como máximo para ejecutar la siguiente instrucción?

Rta: 5 ... 1ro a la TDG, 2do LDT, 3ro PDE, 4to PTE y 5to el Byte en el Page Frame.

¡Son muchos! ¿Como podemos mejorar esto?

En paginación tenemos un cache. En segmentación, por cada selector tengo una parte oculta donde puedo almacenar la información del ultimo selector al que estaba apuntando. Entonces mientras yo no cambie el valor de los registros de segmento, por ejemplo RS o DS, lo cargo la primera vez, pero después mientras no lo cambie, directamente accedo a la parte oculta y me evito este acceso a memoria para obtener la información del descriptor de la GDT o LDT. Me ahorro la segmentación por decirlo de una manera.

A esta parte no le llamo cache porque la cache es una memoria asociativa.

Acá también se guardan los bits de permisos.

Para la Paginación: Intel tiene una cache que no es ni de datos, ni de código. Es una cache de traslación. Esta cache que es una memoria asociativa, entro con los 20 bits superiores de la dirección Lineal, si encuentro una coincidencia, significa que ya hice anteriormente esta doble traslación (ósea ya accedí a esa PDE y PTE), esto tiene asociado la dirección base del Page Frame. Esta cache obviamente tiene capacidad para x cantidad entradas. 4K entradas, puede variar con cada modelo de Intel.

Ingreso con los 20bits de la dirección lineal y si tengo

coincidencia obtengo los primeros 20bits más alto de la dirección del marco de página. La base. Entonces si obtengo esto, le sumo el offset y me ahorro de ir a RAM 2 veces, 2 accesos a memoria.

Según Intel, esta cache de resolución de direcciones o traslaciones de direcciones lineales a físicas tiene un acierto del 95%. Ósea la mayoría de las veces lo encuentra. Esta cache es el TLB. Es una memoria secuencial de acceso rápido, que puede tener la información que busco o no, pero esta información no es de un dato ni de una instrucción, sino que es de una traslación de direcciones.

La mayoría de las veces para acceder a una instrucción o un dato, hago directamente un acceso a memoria. Porque generalmente siempre busco algo que esta contigo ya sea datos o instrucciones, entonces de acá es que tenemos alto grado de aciertos en esta cache.

La parte oculta y la cache están dentro del procesador, por lo cual es mucho más rápido que leer las tablas GDT, LDT, PDE, PTE que están en RAM.

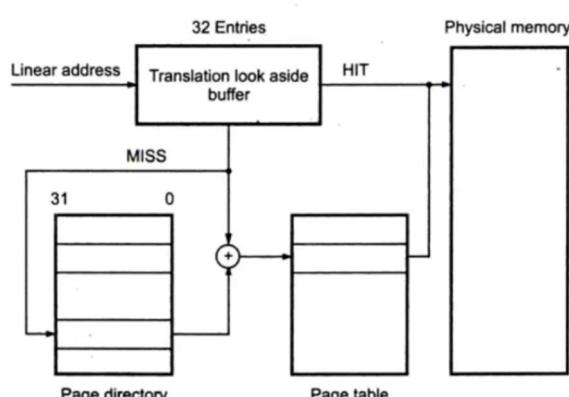


Fig. 4.25 Translation lookaside buffer

según el profe, este grafico confunde. Pero lo importante es que la búsqueda en el TLV y en memoria se realizan en simultaneo/paralelo. Si hay acierto descarta la búsqueda en RAM, si hay fallo, continua la búsqueda en RAM y después actualiza el TLB.

Quedan 2 temas: Niveles de Protección y cambios de Tareas, Ósea que soporte tiene el hardware para proteger que una tarea no rompa a otra, y que soporte de hardware tiene para hacer cambios de tareas rápidamente. Porque cuando veamos SO, vamos a hablar de los cambios de tareas. ¿Como lo hace? ¿Quién lo hace? ... lo hace el operativo, pero tiene ayuda del hardware.

Conclusión:

Objetivo de la 1er traslación es que el procesador crea, por eso es la memoria virtual lo que él cree que direcciona, que tiene espacios de memoria distintos para cada cosa, distintos buses, que nunca se van a chocar o mesclar, esta traslación lo lleva a una memoria lineal.

Si tengo habilitada la paginación en el CR0, esa memoria lineal, probablemente sea mas grande que la memoria física que yo tengo, entonces lo que hago, en vez de cargar todos los segmentos, ósea si un programa se va a ejecutar necesitaría tener en memoria RAM toda la parte de código, el segmento donde están todos los datos, el segmento donde esta la pila, etc., si yo cargo todo eso en la memoria principal, no voy a tener lugar para correr muchos programas a la vez. Entonces alguien dijo, si en vez de cargar todo el programa, cargo justo el pedacito que necesito.

Entonces el Objetivo de la 2da traslación, es hacerle creer de alguna manera al procesador que esos 4Gb de dirección lineal que yo tengo, los tengo en memoria física, pero en realidad podría tener menos. El truco es cargar los pedacitos de programa que necesito para que se ejecute. Si está llena la memoria principal, saco una pagina que no use, libero ese marco, y coloco otra página que necesito porque se va a ejecutar, dentro del mismo marco.

Los algoritmos de remplazo dentro del TLV se tienen que hacer por Hardware, hay que ver que conviene. Si agrandar la capacidad o cambiar el algoritmo.

# -Clase del 24-03-2021 Mecanismos de Protección.

Siempre en modo protegido, en modo real no hay nada, Intel tiene algunos mecanismos de protección tanto para segmentación como para paginación.

En segmentación: hay 3 mecanismos de protección. Y si esta habilitada, un par a nivel paginación.

- 1- Validación de Límite y Base
- 2- Validación de tipo
- 3- Validación por Niveles de privilegio.

¿De qué nos protegemos? ¿Para qué, de quién? Para que, si yo hago un programa, y hago lio ... no perjudique a otro programa de usuario ni al SO. La idea es que queden aislados y no se puedan hacer líos entre programas. En el peor de los casos, mi programa termina por una excepción porque quiso hacer algo que no podía.

**El chequeo de límite** es muy sencillo, si voy a acceder a una posición de memoria, hay un descriptor que me da una base y un límite que me mapea un área de memoria. Entonces este hardware lo que hace es que si la Base + el Offeset si pasa el límite larga una excepción y termina. Para que yo no pueda leer o escribir, o acceder a nada que sea de otro segmento que puede ser de otro programa o usuario.

## El cheo de tipo de segmento.

Table 3-1. Code- and Data-Segment Types

Decimal	Type Field				Descriptor Type	Description
	11 E	10 W	9 W	8 A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Read-Only, expand-down
5	0	1	0	1	Data	Read-Only, expand-down, accessed
6	0	1	1	0	Data	Read/Write, expand-down
7	0	1	1	1	Data	Read/Write, expand-down, accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Execute-Only, conforming
13	1	1	0	1	Code	Execute-Only, conforming, accessed
14	1	1	1	0	Code	Execute/Read-Only, conforming
15	1	1	1	1	Code	Execute/Read-Only, conforming, accessed

Básicamente tengo segmento de 2 tipos (a nivel usuario = No sistema). La pila se considera segmento de datos.

Se compara el tipo del Registro de Segmento con los bits de tipo del descriptor.

Ósea si desde el CS apunto a un descriptor cuyos bits me dicen que mapea un segmento de datos, me va a tirar una excepción.

Otro ejemplo sí, es un segmento de pila, debería estar W=1, lectura escritura, yo hago push y pop. Ósea si direccione un segmento de pila compara que el bit 11 = 0 y w=1, sino larga excepción.

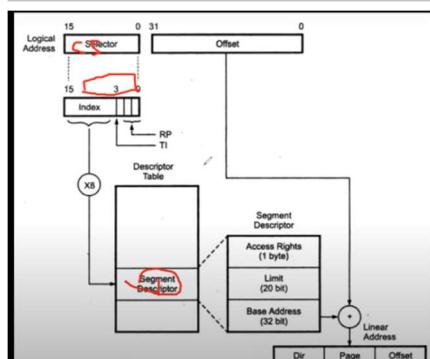
A: es el bit de accedido, es para que el operativo sepa si a sido accedido o no en el ultimo tiempo. Se puede poner a cero periódicamente, y de esa forma sabe el operativo si se utilizó recientemente este descriptor. No da una excepción.

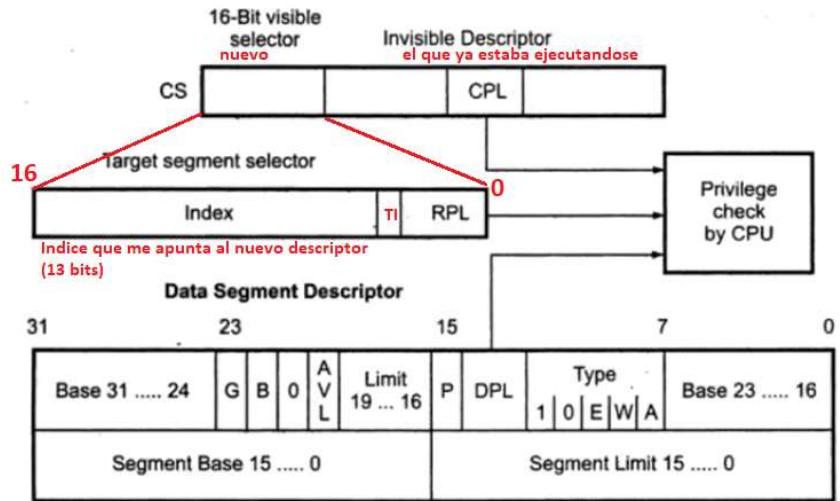
C: bit conformante. Tiene que ver con el mecanismo de protección por segmentación que es por niveles de privilegio. De alguna manera poniendo C =1, podría saltarme realizar algunas validaciones de niveles de privilegio. Está puesto, por lo poco que encontró el profe en el manual, por compatibilidad para atrás con el 80286 que tenía un modo protegido distinto llamado modo virtual.

## Mecanismo por Niveles de Privilegio:

Tenemos en Intel 4 niveles de privilegio, son los 2 bits RP en el selector del registro de segmento. En el nivel 0, el de mayor privilegio, tengo todo el set de instrucciones del procesador disponible. Y en el nivel 3, que es el de menor privilegio, tengo bloqueado todo lo que es entrada – salida, todo lo que es cargar los registros especiales como el GDTR (LDTR), y varias restricciones. A medida que aumenta el privilegio tienen más funcionalidades. La mayoría de los operativos tienen que tener 4 niveles, lo que hacen es usar un nivel 0 donde tienen control de todas las instrucciones y por ende del hardware y un nivel donde tengo muchísimo menos, el 3 en este caso, entonces cuando yo corro el SO, el núcleo del SO trabaja en nivel 0 y las aplicaciones en nivel 3. Evita que las aplicaciones puedan romper otro programa.

¿Cómo hace todo esto? Ver grafica siguiente.





CPL - Current Privilege Level  
RPL - Requestor's Privilege Level  
DPL - Descriptor privilege level

Fig. 4.27 Privilege check for data access

descriptor local o global, que también tiene sus propios bits de privilegio.

Vamos por partes:

Supongamos que mi programa se está ejecutando en un nivel determinado, y de repente le pongo en el Code Segment, coloco un selector para ir a otro segmento de código, pero podre ir siempre y cuando, esté otro segmento tenga un nivel igual o menor de privilegio que donde estoy actualmente. (número más chico)

En el registro de segmento tengo. En la parte visible, el nuevo selector, (índice, indicador de tabla y 2 bits de nivel de privilegio) y en la parte oculta, la información del descriptor apuntado por el selector anterior, ósea el que venia ejecutando actualmente.

Cuando quiero cambiar, cargo el CS con un nuevo valor de 16bits. Los últimos 2 son de privilegio del selector, pero aparte con los 13 del índice apunto a un

Entonces analizando estos 3 valores (CPL, RPL y DPL) determina que va a hacer, si valida y continua o da una excepción y termina.

### Max (CPL, RPL) ≤ DPL

La lógica de la cajita cumple con que, el nivel de privilegio actual y el requerido deberían ser menor o igual numéricamente que el del destino.

Ejemplo del Hola mundo.

Si yo quiero ir directamente de un nivel inferior en privilegio a uno superior, no en número, que sería al revés, daría una excepción. Pero, existe un mecanismo llamado "CALL GATE" (puerta de llamada) que indirectamente me permiten saltar

del nivel 3 a otro, para hacer algo muy puntual y después continuar con la ejecución en el nivel que yo estaba.

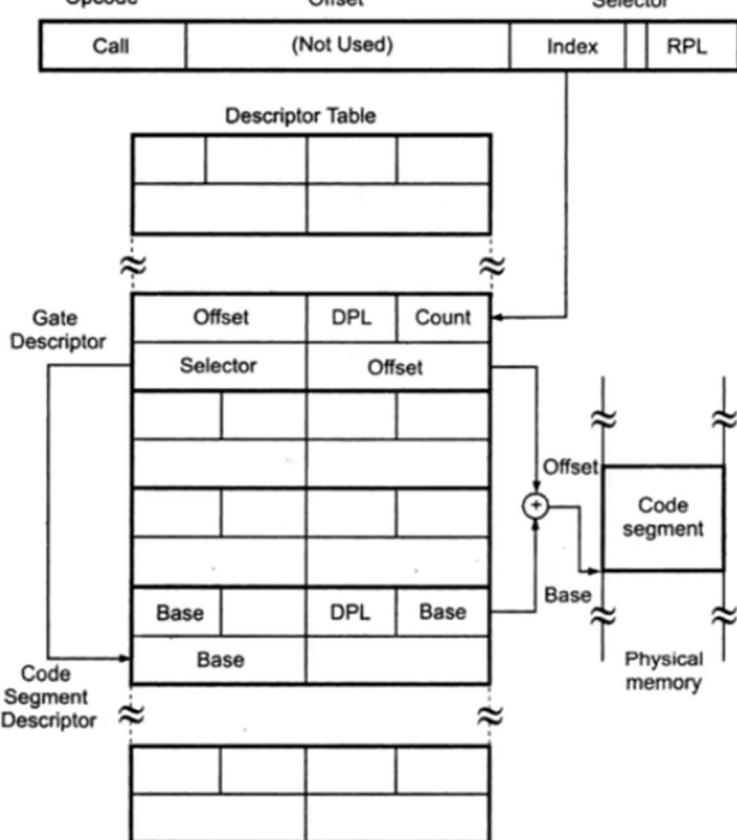
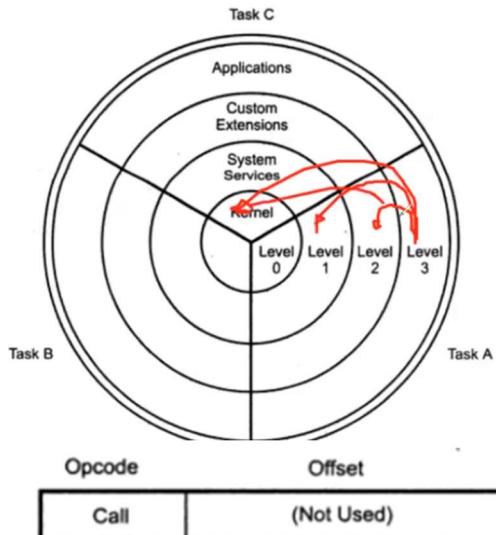


Fig. 4.30 Indirect transfer via call gate

### Funcionamiento de la CALL GATE:

Lo que hago es, en vez de apuntar desde mi selector cargado en un registro de segmento a un descriptor o área a la que no puedo ir, apunto a un descriptor de un CALL GATE. Estos descriptores están en las tablas de descriptores, juntos con los demás, y también ocupan 8 Bytes, pero no me describe o mapea un área de memoria. Tiene otra información, que me permite hacer un direccionamiento indirecto a la memoria a la que yo en realidad quiero ir.

Actúa de interfase entre un programa de un privilegio menor y una función de mayor privilegio, no permite un desplazamiento. Una especie de puntero.

El Opcode + offset era lo que teníamos en el IP si usábamos el CS por ejemplo. O el SS con el EPS.

Coloco el nuevo selector donde yo quiero ir en el RS (CS creo que es siempre para las call gate), esto apunta a un descriptor de gate, el cual tiene el selector del descriptor que apunta al segmento de memoria al que yo realmente quiero ir (Base + Límite), pero no hace como antes que yo usaba el offset que

tenía. Sino que utiliza el offset que estaba en el descriptor de gate.

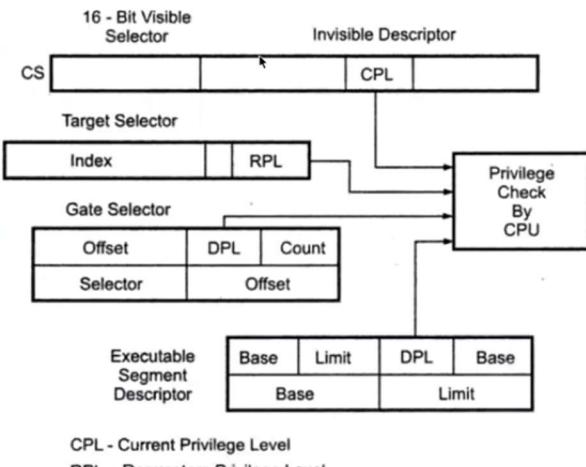
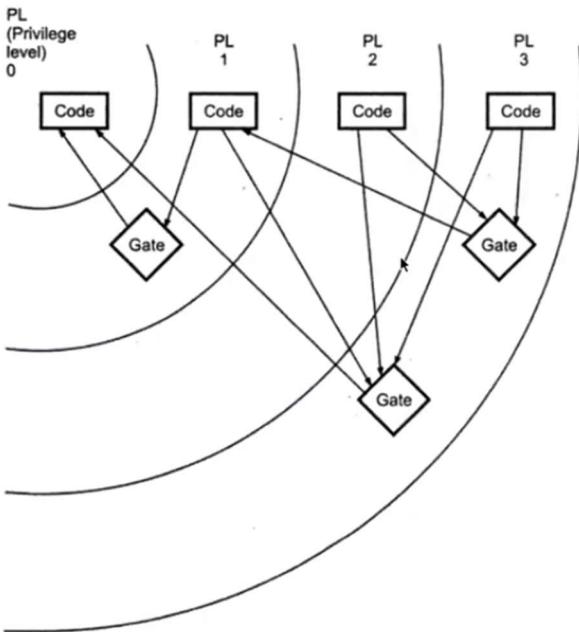


Fig. 4.31 (a) Privilege check via call gate

Entonces yo puedo ir a un segmento de código de otro nivel con mayor privilegio, pero no al offset que yo quiera sino a donde me diga el descriptor de gate. Entonces yo voy a tener un descriptor de gate para distintas funciones / llamadas al sistema. Vamos a tener un montón de descriptores de gate que apunten a distintas rutinas.

Ahora para la validación de niveles, compara el requerido (1er selector) con el del descriptor de Gate, no con el del descriptor destino. Ósea los programas de menor privilegio invocan rutinas de mayor privilegio mediante los call gate.

De esta manera yo puedo ir a un nivel de privilegio mayor, obviamente a ejecutar una rutina que arranca en un punto determinado y una vez que termina hay un procedimiento contrario que me vuelve a mi nivel de privilegio original. Y no puedo hacer otra cosa.



Ejemplo interesante. Analizar.

Acá termina el 3ro de los mecanismos de protección que tiene Intel en Segmentación.

Ahora vienen los mecanismos de paginación si estuviera habilitada.

#### PROTECCION A NIVEL DE PAGINA:

- 1- Restricción del dominio direccionable. **Es el U/S?**
- 2- Chequeo de Tipo. **Es solo el R/W?**

Básicamente lo que hace es, recordando el formato de las entradas al directorio de páginas y a la tabla de páginas a partir de la dirección lineal que contenía la base y otros datos. Tiene un bit U/S que indica si la página es de usuario o de sistema, otro bit que indica si L/E o solo lectura.

Por ejemplo, si yo estoy ejecutando una instrucción en una aplicación nivel 3, menor privilegio, estaría en usuario y no podría acceder a segmentos que fueran de sistema. Y el R/W me marca que si es escritura y yo estoy accediendo a un segmento de solo lectura esto también me va a dar una protección por tipo y me va a dar un error, una excepción. Valida de manera similar a las protecciones por tipo de la segmentación.

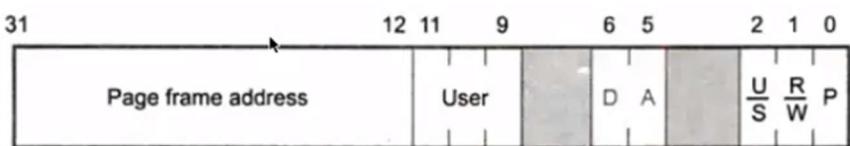


Fig. 4.23 Page table entry

Paginación chequea que se corresponda lo que indican las entradas de directorio y la entrada de la tabla con los descriptores de los que viene la información con la cual se armó la dirección lineal. Verifica si se corresponde con si es de usuario o sistema, si el lectura y escritura o solo lectura.

Extraído de la traducción de Godse.

#### 4.9.5 Protección a nivel de página

Involucra dos tipos de protección:

1. Restricción del dominio direccionable
2. Chequeo de tipo

Los campos U/S y R/W de los PDE y PTE se usan para controlar el acceso a las páginas.

##### 4.9.5.1 Restricción del dominio direccionable

El bit U/S es 0 para el SO (Nivel Supervisor). Cuando el procesador se está ejecutando a este nivel, todas las páginas son direccionables. Si dicho bit es 1, el procesador se ejecuta a nivel de usuario, y solo las páginas que pertenecen al nivel de usuario son direccionables.

##### 4.9.5.2 Chequeo de tipo

A nivel de direccionamiento de página, se definen dos tipos de acceso:

1. Sólo lectura (R/W = 1)
2. Lectura/Escritura (R/W = 0)

Cuando el procesador se está ejecutando nivel de supervisor, todas las páginas se asignan con acceso de lectura/escritura, mientras que a nivel de usuario depende del bit R/W en el PDE y PTE. Si dicho bit es 1, las páginas son de sólo lectura y, si es 0, de lectura/escritura. Cuando el procesador se ejecuta a nivel de usuario, no puede acceder a páginas que pertenecen al nivel de supervisor.

### Último Tema de Arquitectura. Multitarea Multitasking.

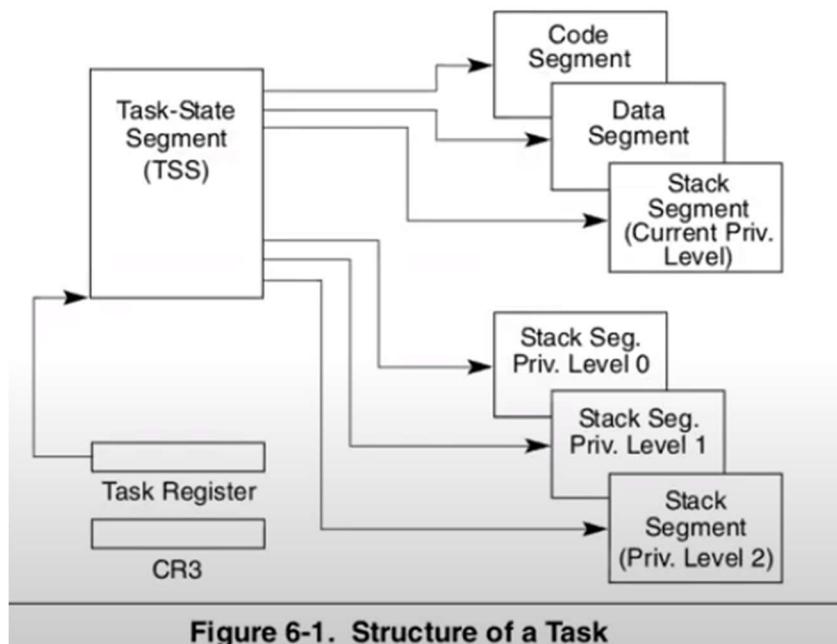
Acá vemos el soporte de hardware que tenemos, porque después cuando se ve sistema operativo se lo ve desde otro punto de vista.

Tanto el manual de Intel como el que se plantean ([suponen](#)), que tenemos soporte para cambiar los datos/valores de los registros internos del procesador rápidamente. Y esto para si yo quiero intercambiar entre tareas.

El concepto de multitarea: supongamos que tenemos un procesador y tenemos varios programas. Si queremos hacer que trabajen a la vez, deberían estar en la memoria principal. Pero en realidad como tengo una sola ALU, una sola CPU, no pueden trabajar a la vez, van a trabajar de trabajar en pseudoparalelo, o nosotros creemos que en simultaneo, pero en realidad no es así. En un momento determinado se ejecuta un programa u otro. Pero no los dos a la vez, de quien es el contenido del IP o los registros internos. No podría mezclarlos. Obviamente hablando de un procesador con un solo núcleo para esta arquitectura.

Ya vimos que, poniendo varios programas en la memoria principal, por los mecanismos de protección no hacen lio entre ellos, y los vamos a ir ejecutando periódicamente. Esto se llama tiempo compartido, le damos la CPU a una tarea por un tiempo determinado, y en algún momento la saco de ejecución a esa tarea y guardo su contexto. Y pongo otra tarea a ejecutarse. El usuario tiene la ilusión de que se ejecutan varias tareas a la vez.

Ya habíamos visto que se puede ejecutar más de una instrucción a la vez del mismo programa. Pero el procesador ejecuta una tarea o programa por vez. Acá vemos que el hardware da facilidades para que el cambio de tarea sea rápido, pero el tiempo que se ejecuta o como lo hace depende del sistema operativo. Hay algunos con prioridades otros de propósito general más balanceados y equitativos, etc.



Generalmente las entra-salida demoran mucho en comparación con un acceso a RAM (nseg), por eso ahí si se justifica el cambio de tareas. (disco rígido, memoria secundaria, en el orden de los mseg). Parece poco, pero si lo vemos en días, sería una tarea demora un día y la otra 3 años. Acá se ve la necesidad de la multitarea.

Entonces de esto se percataron los primeros sistemas operativos entonces vieron como ejecutar más de una tarea a la vez, el problema que tenían era que entre las tareas interactuaban y se podían romper, entonces Intel le agrega protección. Y como era todo un tema cambiar de tarea y ejecutar otra, Intel le agrega soporte de hardware para la multi tarea.

Para el hardware de Intel cada programa que se está ejecutando, cada rutina de interrupción, cada rutina del kernel, a todo Intel le llama tarea. Para el hardware son todas tareas.

Cada tarea tiene un TSS, una porción pequeña de memoria que tiene el estado de la tarea, el contexto, la foto con los valores. Cada tarea tiene un TSS que tiene 104Byte aproximadamente, y se guardan todos los registros internos y algunas otras cosas más.

Bit Map Offset	0000000000000000	T
0000000000000000	LDT	64
0000000000000000	GS	60
0000000000000000	FS	5C
0000000000000000	DS	58
0000000000000000	SS	54
0000000000000000	CS	50
0000000000000000	ES	4C
	EDI	48
	ESI	44
	EBP	40
	ESP	3C
	EBX	38
	EDX	34
	ECX	30
	EAX	28
	EFLAGS	24
	EIP	20
	CR3	1C
0000000000000000	SS2	18
	EIP Editar título	14
0000000000000000	SS1	10
	EIP1	0C
0000000000000000	SS0	8
	EIP0	4
0000000000000000	Back link	0

Fig. 5.1 Task state segment

Como encuentro este segmento TSS en la memoria RAM o en la memoria Lineal. Yo tengo un registro que se llama TR Task Register de 16bits, que tiene el valor (selector) de un descriptor de TSS que esta en la tabla de descriptores globales que apunta y mapea ese estado.

Ósea cada una de las tareas que yo ejecute va a tener un descriptor de estado de tarea TSS y si yo quiero cambiar de tarea, lo que tengo que hacer es cambiar en el TR Task Register el nuevo selector que apunta al descriptor de la tarea a la que quiero cambiar.

Adentro del Segmento de estado de Tarea tenemos los registros internos, el contador del programa, Registros de segmento, todos los segmentos extendidos, algunos registros de control, el valor de 16bits de la LTD, y el back link para volver a una tarea anterior.

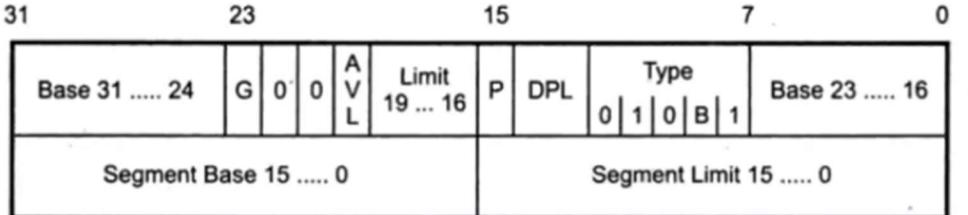


Fig. 5.2 Task state segment descriptor

El TR tiene una parte oculta para no tener que buscar la base y límite de nuevo cada vez que tiene que actualizar los valores del TSS de la tarea en ejecución para realizar un cambio de tarea. Esto agiliza el guardado de los valores, pero igualmente tengo que leer de RAM el TSS de la nueva tarea que voy a ejecutar.

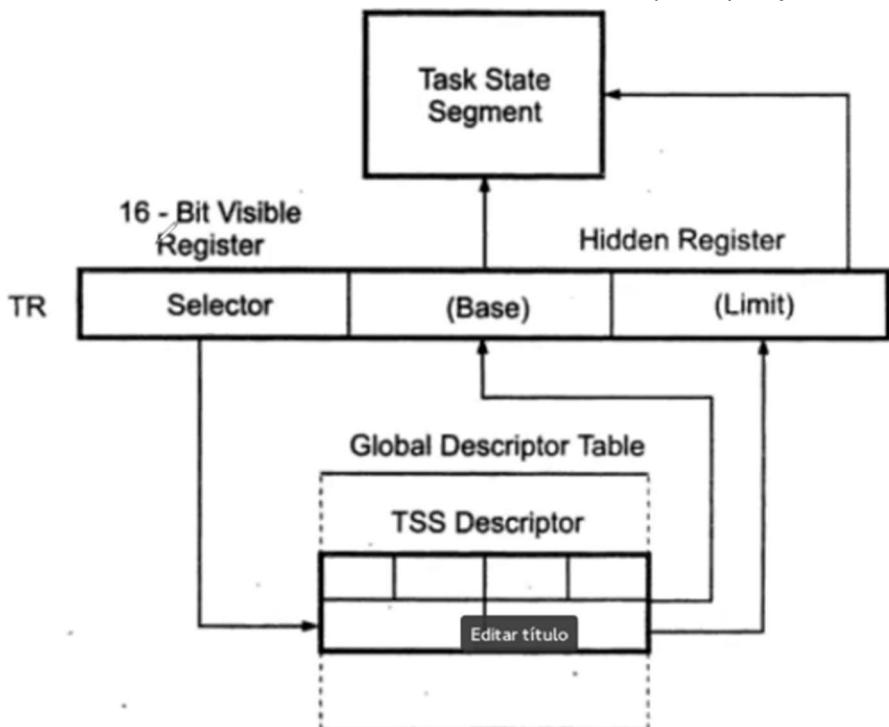
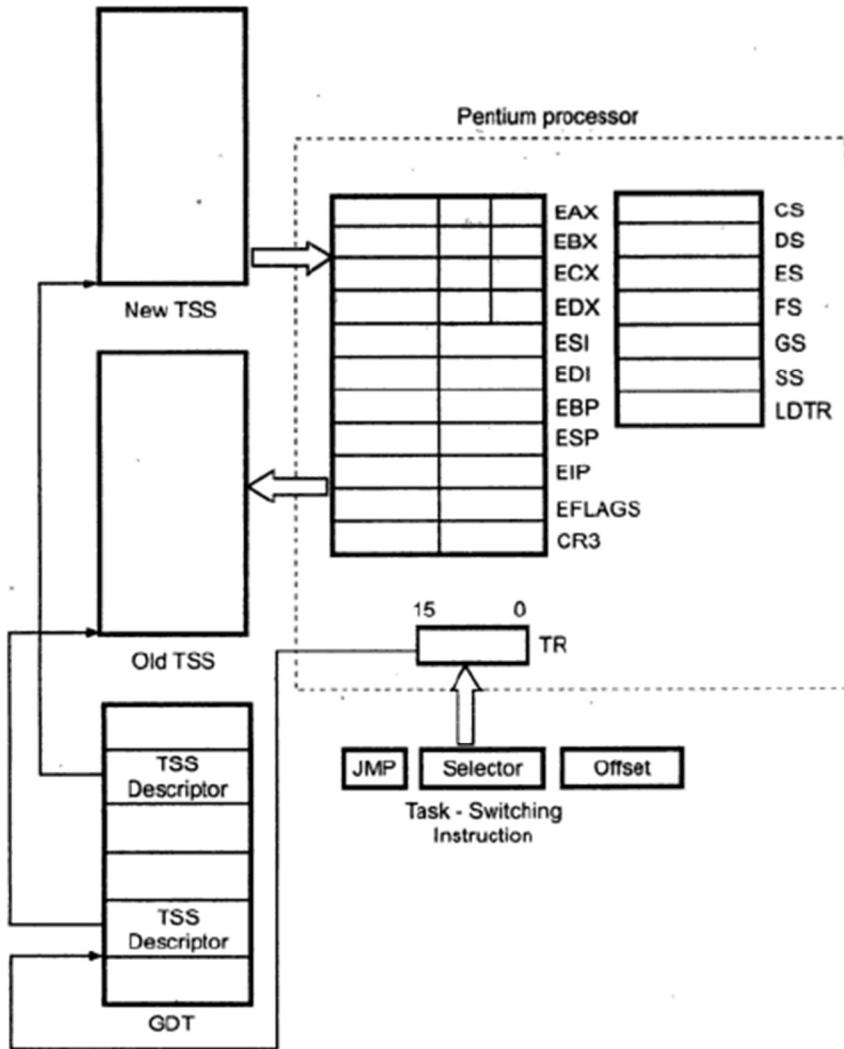


Fig. 5.3 Task register



**Fig. 5.5 Task switch operation**

Se muestra en la grafica como seria un cambio de tarea.

Supongamos que, en un momento dado, se está ejecutando una tarea y el TR está apuntando al descriptor del TSS de la misma. En un momento yo quiero cambiar de tarea, una de las formas es realizando un JMP (poniendo en el offset del jump el selector de otra área). Entonces en el TR ahora voy a tener un selector que apunta al TSS de la nueva tarea.

Antes de realizar el cambio, los registros internos del procesador se guardan en la TTS vieja, después apunto a la nueva tarea modificando el TR, y los valores del nuevo TSS se cargan en los Registros internos del procesador.

Estas instrucciones de guardar y cargar todos los registros internos del procesador a los TSS lo hace Intel directamente, yo no tengo que programar nada.

Estos saltos o cambios de tarea, no los puedo hacer a nivel usuario. Ej.: lo hace el SO cuando le llega una rutina de atención de interrupción de que se le acabo un timer.

Esto es un mecanismo que se hace por hardware.

Hay básicamente 4 maneras de hacer este cambio de contexto o Task Switch dependiendo desde donde viene la necesidad de realizar el cambio. 2 son usando Far Jump o Far Call, esto obviamente el usuario común no lo puede hacer. Hay otra que es indirecta que es cuando me llega una interrupción cambio de tarea y otra cuando termina una rutina de atención de interrupción que es IRET. Estas ultimas 2 no son directas, tengo algo que se llama una Gate Task (puerta de tarea). Es el mismo concepto de Call Gate.

Esto así para que un usuario común que no tiene ni idea que rutina se tiene que ejecutar cuando llega una interrupción, entonces pongo una puerta que se encargue de decir cuál es la nueva tarea que se tiene que ejecutar.

Casos de cambio de tarea:

1er y 2do caso. Acá el operativo decide que una tarea no puede seguir ejecutándose. EJ: La tarea en ejecución realiza una entrada-salida que se la pide al operativo, como va a demorar mucho tiempo, el SO fuerza el cambio.

3ro Atender una interrupción.

4to Termina una rutina de atención a interrupción y vuelve a continuar con otra tarea.

The Pentium processor does task switching in any of four cases :

1. A long jump or call instruction contains a selector which refers to a TSS descriptor. This is the simplest method and can be easily implemented by the operating system kernel at the end of a time slice.
2. The selector in a long jump or call instruction refers to a task gate. In this case the selector for the destination TSS is in the task gate. This indirect method has advantages regarding privilege levels and protection.
3. The interrupt selector refers to a task gate in the interrupt descriptor table. The task gate contains the selector for the new TSS. If the access passes all the privilege level tests, the selector and descriptor for the interrupt task will be loaded into the task register. The nested task (NT) bit in the EFLAGS register will be set.
4. An IRET instruction is executed with the NT bit in the EFLAGS register set. The IRET instruction uses the back link selector in the TSS to return execution to the interrupted task.

El selector apunta a un descriptor TSS. Es más simple.

El selector apunta a un Task Gate. Tiene ventajas respecto a las protecciones y niveles de privilegio.

El selector referencia una Task Gate en la IDT. Se activa una bandera. (NT)

Cuando la bandera esta activada se ejecuta esta instrucción IRET, utiliza el back link para retornar a

la tarea interrumpida. El back link solo se lee cuando se ejecuta un IRET.

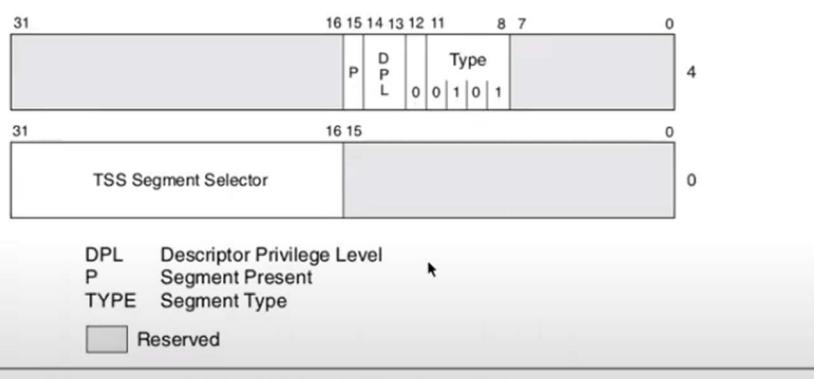


Figure 6-5. Task-Gate Descriptor

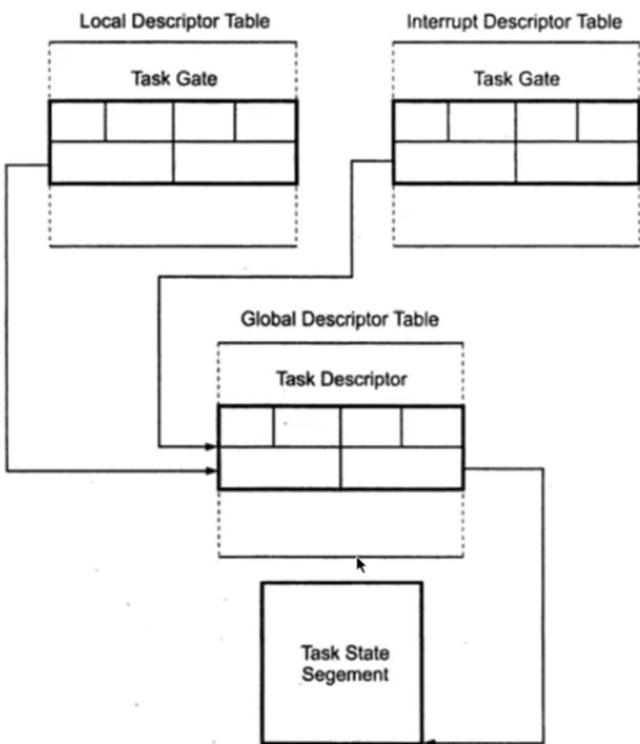


Fig. 5.6 Task switching through task gate

Estos serían los 8 Bytes de un Task Gate Descriptor, que estarían en la GDT. Básicamente lo que tiene 16 bits que apuntan a un TSS.

Se observa que puedo llegar al descriptor de un TSS desde una Task Gate en la tabla de descriptores Locales o desde la tabla de descriptores de Interrupción. Los descriptores de tareas que mapean al TSS están en la tabla global de descriptores.

La idea no es entrar mas en detalle que esto. Sino comprender este mecanismo y que tiene un soporte de hardware que me permite hacer cambios de tarea sin que yo programador tenga que guardar los registros ni cargarlos ni nada por el estilo. En el programa me desentiendo de eso. El SO es el que se va a encargar de realizar los cambios de tareas y eso debido a que cada tarea tiene en la tabla de descriptores globales un segmento de estado de tarea TSS y cuando se realiza un cambio de tareas nosotros ni nos preocupamos de guardar o cargar los registros internos del procesador porque lo hace de forma transparente (oculta) el hardware. (la decisión la toma el SO, pero lo hace el hardware el cambio, no el SO).

TEMAS que se evalúan de arquitectura.

- 1- La estructura básica que habla un poco de los bloques.
- 2- Modo protegido. ¿Qué es? Y el direccionamiento, ósea segmentación y paginación. Las protecciones.
- 3- El soporte para la conmutación de tareas.

De SO la mayor parte se ve del Tanenbaum. Y una parte de otro libro.

# Unidad 2: SISTEMAS OPERATIVOS

- 2.1 - Concepto y definición de un sistema operativo. Evolución histórica, clasificación, system calls y distintos tipos de estructura.
- 2.2 - Gestión de procesos: definición de proceso, estados, jerarquía, inicio y terminación. Implementación de procesos en sistemas operativos multitarea.
- 2.3 - Hilos: definición, necesidad y distintos modelos de implementación: Espacio Usuario y Espacio Kernel. Hilos POSIX.
- 2.4 - Planificador: necesidad y categoría de Planificadores.
  - planificación en sistemas por lotes: FCFS, SJF, SRTN;
  - interactivos: Round Robin, Prioridad, Múltiples colas, SPN, Garantizada, Lotería, Equitativa;
  - Tiempo real: características.
- 2.5 - Comunicación entre procesos, necesidad. Tuberías, FIFO, colas de mensaje POSIX y sockets. Comparativa entre los distintos mecanismos.
- 2.6 - Sincronización: problemas típicos. Herramientas para su solución: señales, semáforos y mutex. Comparativa entre los distintos mecanismos.
- 2.7 - Gestión de la memoria: monoprogramación y multiprogramación sin abstracción de memoria. Abstracción de Memoria: Espacio de direcciones. Multiprogramación con particiones fijas. Reubicación y protección. Intercambio, multiprogramación con particiones variables. Administración de memoria con mapa de bits y con listas enlazadas.
- 2.8 - Memoria virtual. Paginación. Segmentación. Aspecto de diseño e implementación. Algoritmos de sustitución de páginas. Sustitución: de página óptima, de página no usadas recientemente, de página donde la primera que entra es la primera que sale y de página usada menos recientemente usada. Segmentación pura.
- 2.9 - Sistemas operativos tiempo real: Necesidad: Procesamiento secuencial, Sistemas Foreground/Background y Sistemas operativos de tiempo real. Sistemas Operativos de tiempo real: Definición de tareas. El planificador. Tareas y datos. Semáforos. Métodos para proteger recursos compartidos. Colas para comunicar tareas. Rutinas de atención de Interrupciones en RTOS. Gestión del tiempo.

2.1	Sistema operativo	Tanenbaum, Andrew S. <i>Sistemas Operativos Modernos</i> , 3era Edición. Prentice Hall. 2009. <b>Capítulo 1</b> .
2.2	Procesos	Tanenbaum, Andrew S. <i>Sistemas Operativos Modernos</i> , 3era Edición. Prentice Hall. 2009. <b>Sección 2.1</b> . Kerrisk, Michael. <i>The linux programming Interface</i> . 2011. <b>Capítulos 6 y 26, y secciones 24.1, 24.2, 25.1 y 25.2</b> .
2.3	Hilo y planificador	Tanenbaum, Andrew S. <i>Sistemas Operativos Modernos</i> , 3era Edición. Prentice Hall. 2009. <b>Secciones 2.2 y 2.4</b> . Kerrisk, Michael. <i>The linux programming Interface</i> . 2011. <b>Capítulo 29</b> .
2.4	IPC: tuberías, FIFO, cola de mensajes	Kerrisk, Michael. <i>The linux programming Interface</i> . 2011. <b>Capítulos 43, 44, 51, 52 y 53.4</b> .
2.5	Sincronización: mutex, semáforos y señales	Kerrisk, Michael. <i>The linux programming Interface</i> . 2011. <b>Secciones 20.1 a 20.6, 22.1 a 22.7, 22.12, 30.1 y capítulo 53</b> . Downey, Allen. <i>The little book of semaphores</i> , 2nd Ed. Green Tea Press. 2005.
2.6	Gestión de memoria	Tanenbaum, Andrew S. <i>Sistemas Operativos Modernos</i> , 3era Edición. Prentice Hall. 2009. <b>Secciones 3.1, 3.2, 3.3, 3.4 y 3.7</b> .
2.7	Sistemas operativos de tiempo real	José Daniel Muñoz Frías. <i>Sistemas Empotrados en tiempo real</i> , 1ra. Edición. 2009. <b>Secciones 1.1 al 1.7 y 4.1 al 4.9</b> .

## -Clase del 30-03-2021 Sistema Operativo.

### Introducción

No importa mucho la historia. Leer esta clase, no estudiarla creo.

Vemos los componentes o partes de un OS.

Enumerar los distintos tipos. Vamos a trabajar sobre SO de propósito general y después sobre RTOS.

Ver las distintas estructuras que tienen.

La primera parte es solo historia.

### • Temario

- Historia
- Componentes de un OS
- Tipos de OS
- Estructura de un OS

## Historia

1945-1955 Primera Generación (Valvulares)

- Las primeras computadoras no incluían Sistemas Operativos
- Las aplicaciones de usuario se hacían completamente en ISA.
- Gralmente, aplicaciones de cálculo numérico.
- Se ingresaba el programa a ejecutar, y se esperaba una respuesta.
- Problemas ??
  - Tiempos ociosos muy grandes

Estas computadoras que no tenían SO, tenían muchos problemas.

Grandes inconvenientes:

- Consumo y disipación de calor.
- Tamaño
- Confiabilidad un tema importante. (tiempo de vida corto de las válvulas, se rompían a la mitad y había que empezar de nuevo.).
- Para cargar los programas era con clavijas, todo mecánico. Super complicado.

- La mayoría del tiempo estaba calentando el ambiente a la macana, hasta que yo no cargaba el programa no la ponía a correr.

-TIEMPOS OCIOSOS MUY GRANDES y confiabilidad los principales problemas.

## Historia

Konrad Zuse



Imágenes de clase.  
Teclitas para cargar el  
programa y la ALU a  
válvulas.

En los 50s se desarrollan los  
semiconductores y se logra

un poco mas de confiabilidad. Se fue reduciendo el tamaño, el consumo, aumentando la confiabilidad.

En esos momentos el costo de tener una computadora era carísimo (consumo

incluido) y que no hiciera nada no se lo podía permitir. Entonces para acelerar el ingreso de los programas y reducir el tiempo ocioso, lo que hicieron fue un mecanismo de tarjeta perforadas, era mucho mas rápido que con botones o clavijas. Después se podía tener una computadora secundaria que leía las tarjetas perforadas y las pasaba a una unidad de cinta, que era mas rápida.

Las salidas de esos programas eran en impresoras matriciales (onda cortadora de fiambre) como eran lentas, otra alternativa era copiar el resultado en una cinta. Y esa cinta se pasaba a otra computadora secundaria que lo imprimía.

Todo esto permitía aumentar u optimizar el tiempo de uso de la CPU.

## Historia

1955-1965 Segunda Generación (Transist, procesos batch)

- Ejecutaba un job a la vez.
- Sistemas de procesamiento batch de flujo único.
- Los Programas y datos se almacenaban consecutivamente en la cinta o tarjetas perforadas.
- Control de las entradas/salidas y la memoria a cargo del programador.
- Programa entero en memoria.
- Problemas ??
  - Pérdida de tiempo entre un job y otro

Antes todo lo hacia el fabricante básicamente, en esta etapa se empieza a separar las funciones, uno desarrolla el hardware, otro hace los programas, otros la operación (cambiar cintas, tarjetas, etc.). Acá no hacia había ninguna abstracción, todo lo tenía que hacer el programador, tenía que conocer bastante el hardware.

Problema: Seguía ejecutando un programa por vez y cuando terminaba, el operador (persona) tenía que ir a poner otra cinta para que se ejecute y sacar la cinta resultado. Perdía tiempo remplazando las cintas para cambiar de programa. Tuvieron la idea de poner en una misma cinta varias tareas en serie, para que se ejecutaran una detrás de otra y ahí escribieron un mini programa que se llamaba Sistema de Operador, que terminó siendo el Sistema Operativo, que leía todas las tareas una tras otra.

Una persona cargaba todos los programas por tarjeta perforada en una cinta, cuando estaba llena, la pasaba por la CPU y ejecutaba las tareas una tras otra. Esto se llama procesamiento Batch.

## Historia

Unidad de Cinta

Maquina Central



## 1965-1980 Tercera Generación (Circuitos Integrados)

- IBM presentó la familia de computadoras System/360
  - Cálculos & comercial → software compatible
  - Multiprogramación
  - Problemas ?
    - Sistema operativo complejo e ineficiente
  - Aparición de minicomputadoras (p.ej. DEC PDP-7)

Hasta ahora el SO era básicamente una rutina que lo que hacia era leer de una cinta tarea por tarea y lo iba lanzando a la CPU para que se ejecutara y tratar de evitar que la CPU estuviera ociosa.

3er Generación de computadoras. Se desarrollan los Circuitos Integrados. Estas computadoras eran bastante mas potentes, bastante menos consumidoras, mas confiables. El tema que hasta esta época había 2 líneas de desarrollo de computadoras, una que se dedicaban a hacer cálculos tipo científicos o de ingeniería, y otra al tema mas comercial. Ósea unas computadoras mas preparadas para CPU Bound (procesamiento de datos) y otras más preparadas para Entrada-Salida.

IBM tuvo la idea de crear una sola familia de hardware que soporte los 2 tipos de usos, porque hasta ese momento los fabricantes tenían 2 líneas de producto que se encargaban cada uno de una cosa, con hardware distinto y totalmente incompatible. Y como los SO operativos básicos que usaban también eran distintos IBM trato de ponerlo todo en uno solo.

IBM además de unir computadoras orientadas al cálculo y comerciales en un mismo hardware, también usaron un mismo software para la maquina con menos potencia hasta la con mayor potencia. Estandarizaron de alguna manera el uso de un mismo SO para todo.

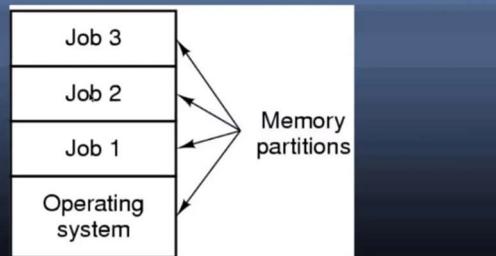
En esta época, ya tenían multi programación, vieron que mientras una tarea estaba esperando entrada/salida estaba ociosa la CPU, entonces vieron la posibilidad de cargar mas de una tarea en memoria, sin que se superpongan o hagan lio entre ellas, pero con soporte de hardware.

Había en esta época discos rígidos, eran caras y necesitaban todavía la pecera con temperatura especiales. Pero el problema, si bien tenían mas aprovechamiento las computadoras, Imaginen los programadores, hacían el programa, de ahí tenia que llegar a alguna unidad de cinta para que lo cargaran, corrían un batch de 20 o 30 tareas que se ejecutaba en algún momento, me traían el resultado, y decía "error en la línea 12, se olvidó un punto y coma jajajaja".

PROBLEMA: era el tiempo en que el programador quería correr su aplicación y obtenía efectivamente un resultado.

## Historia de Sistemas Operativos

### Multiprogramming



El software del SO si bien era compatible para hardware para calculo y comercial y para poca y mucha potencia, este SO era bastante complejo e ineficiente porque tenía muchas funcionalidades para las computadoras de gama alta de estas 2 aplicaciones, pero si quería correr el mismo software en una de gama baja era bastante complejo.

Todas estas computadoras eran main frame.

Multiprogramación: nace el concepto.

La idea es poner en la misma memoria RAM varias tareas, entonces cuando una se está ejecutando, y necesita entrada salida, puedo con ayuda de este software (el SO) y un soporte de hardware, que se ejecute otra de las tareas.

## Historia de Sistemas Operativos

### Multiprogramming

protección entre particiones .... mejor por hardware

Spool (Simultaneous Peripheral operation on line) no mas cintas ....

Problema ??

- Tiempo perdido del programador

multiprogramming + terminales = time sharing (primer OS CTSS)

Generar respuesta dentro de un período de tiempo limitado.

MIT, GE y Bell usaron el sistema CTSS para desarrollar su propio sucesor, Multics. (MULTI Information Computing Service)

Luego del fracaso comercial de Multics, en los Labs Bell Thomson y Richie escriben UNIX PDP-7

Protocolo estándar de comunicaciones TCP/IP ambientes militares y universitarios - 1975 (ARPANET)

Spool: La CPU permitía leer de tarjeta perforada, o escribir en una impresora sin esperar que el programa termine.

Si bien la multiprogramación mejora, seguía existiendo el Delay grande para los programadores que querían compilar el resultado

Todo esto eran SO muy primitivos. No escribí más de la historia. Hablas de terminales BOBA. Multiplexaba los recursos de la CPU en el tiempo. Mejora la atención a los programadores.

La idea era vender servicios, ellos tenían una computadora muy grande (servidor), y el que quisiera servicio de procesamiento le ponían una terminal y le cobraban por el tiempo de uso. (cómo funciona la nube hoy en día). Las terminales no tenían poder de

## Historia

1980-Actualidad Cuarta Generación (PC computers , LSI)

- 8080 nace en 1975
- IBM PC 1980
- CP/M
- Microsoft
- Apple Lisa → Apple Macintosh
- Windows
- Network Operating Systems
- Modelo de Computación Cliente/servidor
- Aparece el software fuente abierto y libre
- Se funda Open Source Initiative (OSI) para beneficios futuros de programación open-source.

cómputo, vos hacías algo, se ejecutaba en el sistema central y después te daba el resultado.

Crean el lenguaje C, para no tener que programar lo mismo de nuevo para cada modelo de computadora porque cambiaban los sets de instrucciones de una a otra, incluso en el mismo fabricante. Despues había que hacer un compilador para cada hardware distinto, pero era mucho menos trabajo que reescribir el SO.

Hasta este momento IBM apostaba a las Main Frame, y acá larga las computadoras más chicas, personales.

Mas historia que no anote.

La 5ta generación seria las computadoras cuánticas.

# Sistema Operativo.

## Sistema Operativo

¿Entonces, que es un Sistema Operativo ?

Existen dos enfoques

- Máquina extendida
  - Capa de abstracción
    - Uso de Syscalls por ejemplo
- Manejador de Recursos
  - Procesos (que es un proceso?)
  - Memoria
  - I/O
  - Sistemas de Archivos
  - Multiplexación (Compartir recursos en tiempo y espacio)

¿Qué es? Sería como una capa, proporciona una interfaz más amigable entre el hardware y el usuario.

Ej: Si no existe el SO. Yo usuario escribo un programa que lea o escriba algo en un disco, memoria secundaria, yo debería conocer cual es el dispositivo periférico que está conectado y como programarlo, pero si mañana cambio de PC y en vez de tener un disco IDE tengo un SATA o USB, debería cambiar mi programa.

La Idea es que el SO trata de darme es una interfaz genérica y única, para lo que yo quiera hacer. No tener que cambiar mi programa cada vez que se cambia algo del hardware.

El SO lo que hace es brindar una interfaz más genérica y de alto nivel, y más homogénea para los dispositivos de bajo nivel. Este es uno de los enfoques y se llama Maquina Extendida. Me da una abstracción de todo el

Hardware Feo. El SO para abajo tiene interfaces feas con el hardware y para arriba lindas con el usuario digamos.

¿Como accedo al SO? Usando llamadas a sistema. Cuando hago una llamada a sistema para leer o escribir no necesito saber que tipo de dispositivo es.

El otro enfoque es lo que se llama un manejador de recursos. Donde la computadora tiene muchos recursos y lo que tengo que hacer es compartirlo entre los distintos usuarios.

Pero ....¿Que es un Sistema Operativo ?

No hay definición completamente adecuada:

- El objetivo de las computadoras es resolver problemas del usuario (son muy variados)
- Todos los programas usan operaciones comunes (ej:E/S)
- El software que hace operaciones de control de Hw.
- Aquel programa que brinda un entorno para que se ejecuten otros programas, generalmente se ejecuta permanentemente (kernel)

No vamos a encontrar una definición completa, para arrancar el SO es un programa que le brinda a otros programas un entorno para que les sea más fácil trabajar.

Según Tanenbaum básicamente la función del SO es hacerle más fácil la vida al programador con abstracciones, o con una maquina extendida se dice, sería un set de instrucciones de más alto nivel por eso extendida.

Y la otra función multiplexar los recursos en todas las tareas que

quiero realizar.

# Gestiones del Sistema Operativo

## Gestión de Procesos:

Crear/terminar/suspender/reanudar procesos

## Gestión de Memoria:

Controlar que partes de memoria corresponden a cada proceso

Asignar/liberar espacio de memoria a los procesos.

## Gestión de Almacenamiento:

Archivo y directorio (almacenamiento lógico)

Creación/borrado archivos y directorios

Asignación de archivos/directorios a medios secundarios

## Protección y Seguridad:

Evitar sobreescrituras y monopolización de CPU

Esquema de usuarios y permisos de acceso

¿Qué hace básicamente el sistema operativo o que controla? ¿O que es lo que se encarga de manejar?

## Tipos de Sistemas Operativos

Sistemas de propósito general: Sistemas para pc.

Sistemas de tiempo real: Tareas muy específicas, requisitos rígidos de respuesta en tiempo.

Sistemas Multimedia: Posibilidad de trabajar video/sonido. Algunas restricciones de tiempo.

Sistemas de mano (PDA): Procesamiento lento para ahorro de energía. Pocos recursos HW.

Sistemas Centralizados: Todo procesamiento en servidor, terminales "bobas"

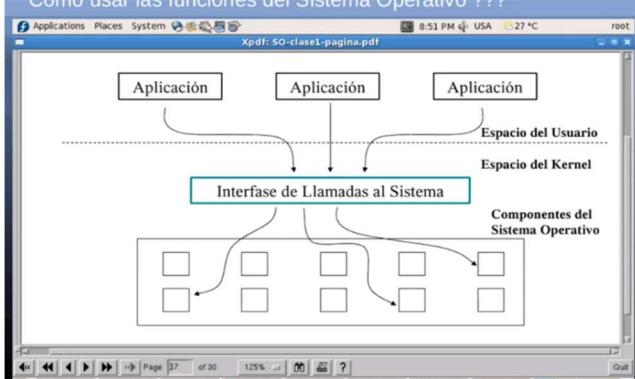
Sistemas Cliente Servidor: La presentación se hace en el cliente.

Sistemas entre Iguales (Peer to Peer): Cliente/servidor simultáneamente. Previa Registración.

Sistemas Basados en WEB: Centralizado ? Uso de alta disponibilidad y equilibrio de carga.

## Estructura de un Sistema Operativo

Como usar las funciones del Sistema Operativo ???



No le da mucha bola a esto el profe.

¿Como hace el programador para utilizar las funcionalidades del SO, eso de abstraerme de las interfaces feas con el Hardware? Mediante las llamadas a sistema

YO lo que hago es una aplicación, donde realizo una llamada a sistema y el SO es el que se encarga de lidiar con el Hardware. El SO tiene componentes, dentro de los cuales están los drivers, los manejadores de tal o cual controlador de hardware. Y yo lo único que hago es decir SO encárgate de hacer esto, y el dependiendo del hardware que tenga es lo que va a hacer.

Tanenbaum toma un ejemplo de llamada al Sistema, READ.

```
NAME
    read - read from a file descriptor
SYNOPSIS
    #include <unistd.h>
    ssize_t read(int fd, void *buf, size_t count);
```

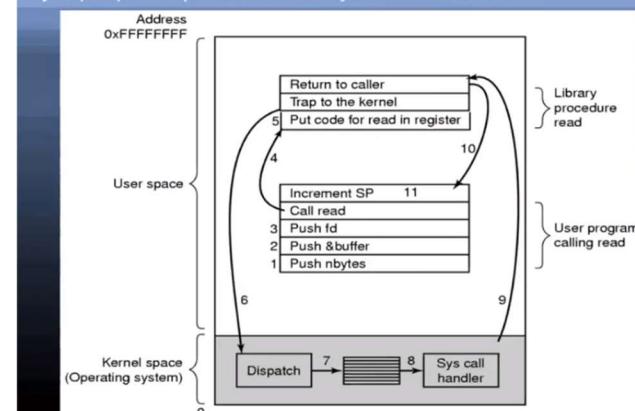
En este ejemplo, tengo un nivel de usuario 3, con el set de instrucciones reducido, sin E/S.

Para ejecutar la instrucción read envió 3 argumentos, y como suponemos que los pushea a una pila, están en orden inverso.

Suponemos que tenemos mi programa en c, python o cualquier lenguaje. Que esta corriendo, y cuando ejecuto read, paso los argumentos y hago un call a un procedimiento de biblioteca, el cual hace efectivamente la llamada a sistema. Que parecida a una función común, pero tiene 2 temas, 1ro pasa de modo usuario a kernel. 2do no va a cualquier lado, no puedo decirle a donde, sino que va a un punto específico. Una vez que el call gate programa el periférico para leer, el CPU después sigue con otra cosa. Mi programa se bloquea, no se sigue ejecutando, en algún momento ese periférico que yo programé me va a enviar una interrupción de hardware diciendo, "che! Ya están los datos que pediste". En ese momento el manejador de interrupciones, va a ver que llegó una interrupción de ese periférico y va a saber que es para el programa que yo

quería, y recién ahí va a retornar a modo usuario a la función siguiente al call read. Desapila los argumentos. (Linux según vimos con taffe pone los argumentos en registros específicos, no en la pila)

Ejemplo pasos para hacer un syscall de lectura



```

#include <unistd.h>
#include <stdio.h>

int main (){
    char buff[100];
    int leido;
//  scanf("%s", buff);
//  printf("leido del teclado: %s \n", buff);
    leido = read(0, buff , 40);
    write (1,buff, 20);
    return 0;
}

```



Estos descriptores que pasamos como argumentos, no tienen nada que ver con los anteriores, y son un numero entero que esta asociado con un canal. Cuando arranca un proceso a ejecutarse, por defecto tienen 3 canales

abiertos, 1 para el teclado y 2 de pantalla. 0 teclado. 1 pantalla, 2 error por pantalla.

Conclusión: Antes de realizar la llamada a sistema, tengo que pushear las cosas en la pila, en Linux lo pone aparentemente en registros específicos, después llama a la biblioteca que realiza el call gate, una vez que termina retorna a la biblioteca, de ahí retorna a la función, y esto (SP) saca las cosas de la pila.

Ahora viene como estanjeramente estructurados los SO. Hay varias formas.

## Estructura de un Sistema Operativo

Comenta algunos.

- Monolítico
- En capas
- Microkernel
- Máquinas Virtuales
- Microkernel
- Cliente Servidor
- 
- Monolítico
  - Bien definida las interfaces entre las rutinas
  - cualquier rutina puede llamar y ver a cualquier otra.
  - Estos componentes son independientes entre si
  - Soporte de extensiones :
    - dll
    - modulos

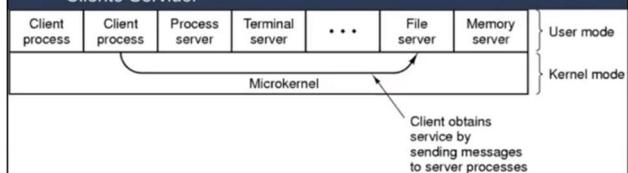
- En Capas
  - Capas Gerárquicas, cada uno construida sobre la anterior.
  - Cada capa se comunica con la adyacente
  - Menor throughput que los kernels monolíticos.

### Estructura de Sistema Operativo "THE" Dijkstra, Multics

Layer	Function
5	The operator
4	User programs
3	Input/output management
2	Operator-process communication
1	Memory and drum management
0	Processor allocation and multiprogramming

- Microkernel
  - El kernel del S.O. provee un reducido número de servicios
  - Se lleva gran parte de las funciones del Kernel a capas superiores
  - Microsoft (hibrido) y Minix

### Cliente Servidor



Monolítico: son todas funciones y se compilan, y es un solo binario que se está ejecutando. Igual se podría organizar por servicios, generalmente hay una sola función principal, después esa invoca a los distintos servicios y por ahí hay también aplicaciones utilitarias. Pero básicamente es toda una sola cosa.

Pro: Desde cualquier servicio podría utilizar cualquier utilidad, o desde el main invocar cualquier servicio sin ninguna limitación. Todos los servicios se ven contra todos los serv.

La contra: que tendría que tener bien clara las interfaces porque podría romper todo.

Linux es monolítico, pero tiene soporte de extensiones. Que me permite poner lo mínimo y a medida que el operativo necesita algún driver de lo que sea, lo lleva a la memoria, sino no lo usa.

La mayoría de los Unix son monolíticos. VENTAJA: Muy Rápido.

En Capas: no se han visto nuevos, arma capas donde cada una le da un servicio a la anterior. Ej: la capa 0 solo hace multiplexación de tareas en la cpu, la capa 1 lo único que se encarga es de asignar memoria, la 2 se encarga de IPC, la 3 entrada salida, la 4 programas de usuario, 5 interface con el operador.

Básicamente una va sirviéndose de lo que ofrece la anterior, esta buena, son jerárquica, el tema es que es mucho mas lento. Es prolífico, pero no se usa

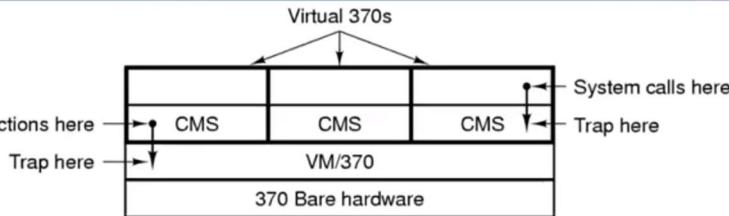
Microkernel: básicamente lo que hace es que solo las funcionalidades principales las deja en modo kernel, el resto de las cosas, ej: manejo de sistema de archivos, manejo de memoria, creación de procesos, todo ese tipo de cosas las hace un programa en modo usuario.

Esto es tratando de tener más confiabilidad. Parecería que tuviera menos seguridad.

Microsoft tiene algo de esto, pero es un híbrido.

-Cliente Servidor: muy parecido, funciona de la misma manera, la diferencia es que los distintos procesos en modo usuario están en distintas máquinas, una especie de SO distribuido. Un solo microkernel, pero los servicios están en otras máquinas.

- Máquinas Virtuales
  - Un sistema de time sharing provee:
    - multiprogramación
    - máquina extendida
  - La esencia es separar completamente esas dos funciones.
- Estructura de VM/370 con CMS



Máquina virtual: básicamente se basa en el concepto que hablamos donde yo puedo multiplexar el hardware y hago multiprogramación y por otro lado tengo una máquina extendida donde tengo una abstracción de las interfaces feas del hardware.

Lo que vio IBM en los 70s, es que, si separo esas 2 funciones totalmente, se podría hacer un SO que solo se encargue de la multi programación.

VN/370 se encarga de poder correr varios programas simultáneamente. Aísla los programas entre si, y como no tienen una máquina extendida, los programas creen que están interactuando con el hardware, porque no tengo ninguna abstracción.

Entonces el Truco, es que el SO realiza multiplexación, y después en vez de poner distintos procesos, instala distintos SO CMS. Estos creen que hablan con el hardware, pero en realidad hablan con la capa VM/370. Que es la que interactúa con el hardware.

### • Máquinas Virtuales en la actualidad

- Necesidades
- Soporte de hw
- Ejemplos Xen, VMWare, KVM, docker (no está en el gráfico)

VM hoy en día se llama hipervisor.

Se usa para ahorrar Hardware, configuras un servicio en un SO distinto.

Sigue hablando de esto, no se para que.

Docker no es una virtualización a nivel SO, sino a nivel aplicación. Yo tengo una máquina y lo que hago es, empaquetar las aplicaciones con bibliotecas, la ventaja es que me ahorro de usar varias capas, la desventaja que puedo virtualizar varias máquinas, pero todas para el

mismo SO.

No anote mucho de esta última parte, porque no es relevante creo.

## -Clase del 30-03-2021 Procesos y Señales.

### FUNCIONES PRINCIPALES DE UN SISTEMA OPERATIVO

- PROCESOS:
  - Necesidad y Definición
  - Creación/Terminación de procesos
  - Estados de procesos / Cambios
  - Jerarquía de procesos
  - Implementación de procesos en Linux
- SEÑALES
  - Distintos tipos de Señales
  - Manejador
  - Implementación de señales en Linux

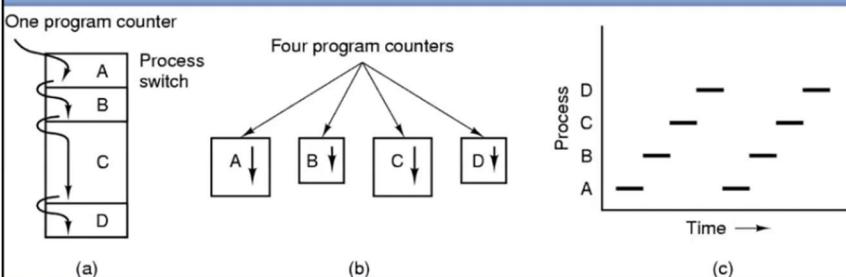
¿Porque quiero más de un programa ejecutándose? Básicamente porque la velocidad del procesador comparada con la entrada-salida es mucho mas rápida, y si un programa lo pongo a ejecutar y esta único y los mando a hacer una entrada salida, el procesador esta ocioso mucho tiempo.

### Procesos

#### ¿Necesidad de multiprocesamiento?

- Velocidad Procesador
- Velocidad Entrada Salida
- Pseudoparalelismo
  - Un solo procesador
  - Una instrucción por vez
  - ¿Cómo lo logro?

## Pseudoparalelismo



Programas DEBEN estar en memoria

Asignación de tiempos (Problemas con Real Time)

No programar basandose en supuestos de tiempo

Si se ejecuta nuevamente probablemente terminarán en distinto orden.

## Procesos cont.

¿Qué es un proceso?

Programa (entidad pasiva) en ejecución

+ valor del contador de programa

+ valor de los registros

+ pila del proceso (parámetros de funciones, direcciones de retorno, etc.)

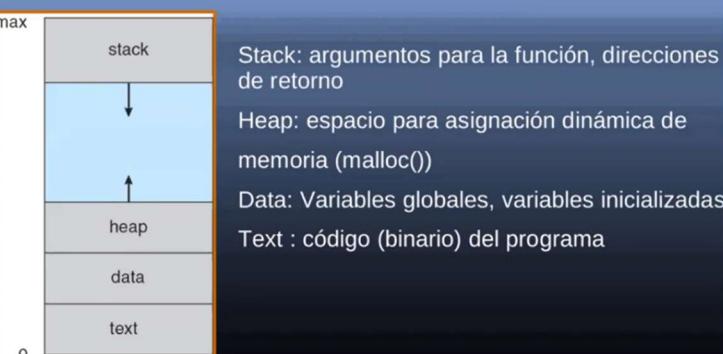
### PROCESO (entidad activa)

que lo pone en pausa es el SO cuando cree conveniente, no yo programador.

EL proceso es una primera abstracción que creamos y nos permite la comutación de programas, procesos.

Analogía: Supongamos una pizzería, donde el cocinero tiene la receta. El cocinero es la CPU. Tiene que recordar si le puso sal.

Ubicación del proceso en memoria:



Supongamos que tengo un proceso en ejecución, cuando empieza el tengo que asignar todos los registros de Segmento, SC, DS, SS, etc. y en el espacio virtual del proceso normalmente el SC se le llama área de texto donde esta el binario o lenguaje de maquina que se va a ejecutar, datos donde tengo las variables estáticas. Y 2 segmentos que varían en tamaño durante la ejecución.

¿Que guarda la pila? ¿Porque varia? Esta pila o Stack la manejo yo programador, no es el TSS que maneja el SO cuando realiza el intercambio de tareas que es invisible para el programador. Esta

pila se utiliza internamente cuando el programa mismo llama a funciones o rutinas. (anidadas o no)

Si creo hijos, cada proceso tiene un área de memoria aislada del resto, por las protecciones, no de niveles, pero si de límite y tipo de seg. **¿Qué guarda Heap?**

## Creación de Procesos

¿Para qué creo procesos?

- Inicialización del Sistema
  - Init o systemd
  - Servicios
    - Background, foreground (bg, fg, &, at)
- Proceso hace llamada a sistema "fork()"
  - Crea un nuevo proceso ... copia...
- Desde Interpretador de comandos (o doble "click")
  - Para sistemas interactivos
- Nuevos trabajos batch
  - Para sistemas de procesamiento por lotes

¿Cuándo se crean, porque crean y para qué se crean procesos?

**1ra -Cuando arranca el sistema**, tengo varios procesos que se crean, conocidos como servicios, daemons (demonios mal traducidos) monitores y hacen alguna tarea. **De manera automatizada sin requerir atención de mi parte.**

Programa con los cuales no tengo interacción, pero se arrancan al inicio del sistema ej: chequea si llego un correo, revisa si hay algún virus periódicamente, etc.

En UNIX se llama init o systemd el primero que se ejecuta.

Background son los procesos en 2do plano, que no tienen generalmente asociado una entrada salida

¿Para qué creo procesos?

- Inicialización del Sistema
  - Init o systemd
  - Servicios
- Background, foreground (bg, fg, &, at)

Foreground son con los que nosotros interactuamos.

**2da Fork(): Es una llamada a sistema que crea una copia de mi programa.**  
Para que, si yo escribo un programa, ¿ahí adentro digo crea otro

programa? Rta: Para realizar una tarea específica.  
Gano en velocidad. Mientras una espera puedo ir realizando las otras. Entonces de alguna manera, cuando yo tengo tareas que las puedo dividir en partes, y puedo implementarlas en distintos programas y trabajan de manera colaborativa, me sirve porque la principal se ejecuta mucho más rápido.

**3ra Cuando quiero hacer algo nuevo:** ejecuto un comando o abro un programa, eso crea una tarea o un nuevo proceso.

**4ta nuevo conjunto de tareas.**



Cuando se crea un hijo se copia todos los segmentos de memoria y también los registros, ósea que, debido al contador del programa, el hijo se ejecuta a partir de la siguiente línea en la cual fue creado.

El orden de ejecución depende del planificador, y no siempre es el mismo. Esto no es determinístico, por eso no puedo hacer supuestos de tiempo u orden, salvo que use herramientas o mecanismos que me lo permitan.

La llamada a sistema fork retorna el PID del hijo al padre. PID: identifica a cada proceso y es distinto. Número positivo entero mayor a 1. Y al hijo que se empieza a ejecutar en ese mismo punto, le retorna 0.

Echo \$? : me dice como terminó el último proceso que ejecuté. (Exit status)

Normalmente el Exit status sirve para dejar cosas programadas para que se ejecuten en algún momento determinado cuando yo no esté.

Normalmente cuando algo termina bien por convención retorna con 0. Si hay error con otro valor.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    int pid;
    printf("soy el padre\n");
    pid = fork();
    //hijo
    if (pid == 0){
        printf("soy el hijo\n");
        return 0;
    }
    usleep(200);
    printf("cree el hijo\n");
    return 0;
}
```

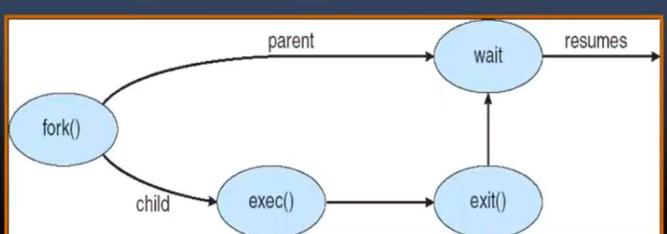
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    int pid;
    printf("soy el padre\n");
    pid = fork();
    //hijo
    if (pid == 0){
        sleep(1);
        printf("soy el hijo\n");
        return 0;
    }
    wait(NULL);
    printf("cree el hijo\n");
    return 0;
}
```

## Creación de Procesos con fork()

Alternativas Padre:

- Ejecuta concurrentemente con hijo
- Espera que finalize el hijo
- Alternativas Hijo:
  - Usa mismo programa y datos que heredó padre
  - Carga un nuevo programa



El proceso padre tiene 2 alternativas, esperar que finalice el hijo o no.

Al wait() se le puede pasar de argumento un puntero para almacenar el valor de retorno del hijo.

Ej: El padre asigna un hijo para atender a cada cliente, y no los espera, sigue con el siguiente cliente.

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    int pid;
    printf("soy el padre pid = %d y mi padre es %d \n", getpid(), getppid());
    pid = fork();
    //hijo
    if (pid == 0){
        sleep(1);
        printf("soy el hijo\n");
        return 0;
    }
    wait(NULL);
    printf("cree el hijo\n");

    return 0;
}

```

```

carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ./fork1
soy el padre pid = 424236 y mi padre es 388229
soy el hijo
cree el hijo
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ./fork1
soy el padre pid = 424243 y mi padre es 388229
soy el hijo
cree el hijo
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ps
  PID TTY      TIME CMD
388229 pts/4    00:00:00 bash
424261 pts/4    00:00:00 ps

```

Hay una llamada al sistema que se llama getpid

Una alternativa es que el hijo vaya preguntando con los if, pero debería estar en el mismo código del padre. Con cada fork todo el código se duplica y uso de manera ineficiente la memoria.

Una alternativa es si ejecuto exec() y le paso como argumento que binario quiero, lo que hace es remplazar el segmento de código del hijo. Sobrescribe el código que era copia del padre y reubica el contador de programa al principio.

Exec pone el el cuerpo del proceso hijo otra receta, otro binario.

Los programas no son eternos, tienen que terminar una vez que hacen lo que necesitamos que hagan. Hay una terminación normal, cuando ejecuto una tarea le coloco exit() o return 0. Y después si desde el SO realizo un "echo \$"? Me dice como retorno.

Una terminación voluntaria: el exit status retorna distinto de 0 y lo puedo consultar con el echo\$?

## Terminación de Procesos

### Tipos de terminacion

- Terminación Normal
  - Tareas que realiza el SO → exit()
  - echo \$?
- Terminación con Error Voluntaria
  - exit(!=0) echo \$?
  - standard error
- Terminación con Error Involuntaria
  - No hay mas memoria
  - Violación de Segmento
  - División por cero
- Recepción de una señal (Que es una señal?)
  - Comando kill, llamada a sistema "kill()"
  - Handler de señales, llamada a sistema "signal()"

Ejemplo de terminar bien y con error de manera voluntaria.

```

carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ls
Clase2.odp  fatal  fork1  fork2  fork3  fork4  fork5  senial0.c  senial2.c  senial4.c
Clase2.pdf  fatal.c fork1.c fork2.c fork3.c fork4.c fork5.c  senial1.c  senial3.c  senial5.c
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ echo $?
0
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ls pepe
ls: cannot access 'pepe': No such file or directory
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ echo $?
2

```

Ejemplo de error involuntario. El proceso termina, pero lo hace involuntariamente. El SO, cuando el proceso quiere hacer algo que no puede hacer, lo termina.

```
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ./fatal
Soy un proceso que terminara involuntariamente
Floating point exception (core dumped)
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ echo $?
136
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
    printf("Soy un proceso que terminara involuntariamente \n");
    printf ("operacion %d\n", 100/0);
    return 0;
}
```

Error división por cero.

Otra forma de terminar los procesos es cuando reciben una señal. ¿Qué es una señal? Es una interrupción por software, no un hardware que ejecuta una interrupción. Funciona parecido.

Si a un proceso le llega una señal que es un evento de software, lo que hace es ejecuta una rutina de atención de esa señal. Por defecto cada señal, porque son un montón, tiene un comportamiento por defecto, que yo lo podría modificar si quiero.

```
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ kill -l
 1) SIGHUP      2) SIGINT      3) SIGQUIT      4) SIGILL      5) SIGTRAP
 6) SIGABRT     7) SIGBUS      8) SIGFPE      9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Ejemplo: copia el contenido del archivo fork1.c y crea el archivo termina.c pegando dentro los datos copiados. Luego la abre con vim, lo edita y lo compila con make.

```
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ cp fork1.c termina.c
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ vim termina.c
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ make termina
```

```
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ vim termina.c
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ make termina
cc    termina.c -o termina
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ./termina
soy el padre pid = 424673 y mi padre es 388229
Killed
```

Tengo un proceso que se está ejecutando, no hace nada. Pero le envío una señal para que termine.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    int pid;
    printf("soy el padre pid = %d y mi padre es %d \n", getpid(), getppid());
    sleep(100);

    return 0;
}
```

```
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ kill -9 424673
Clase2$
```

La señal la mando desde otra pestaña/consola. Puse la señal (-9) y el PID del proceso.

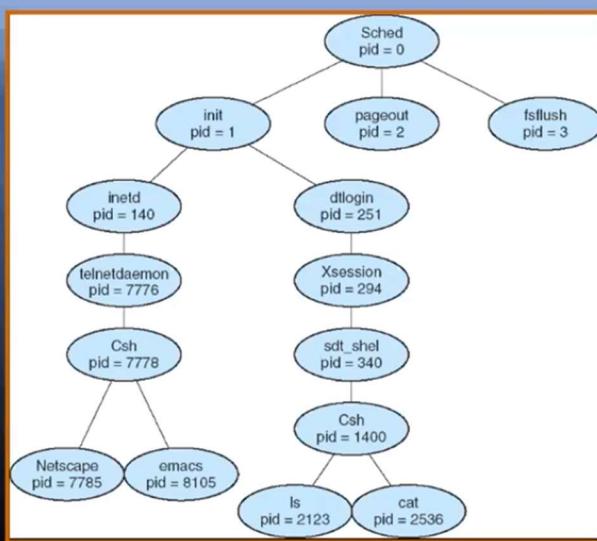
```
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ kill -9 424673  
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$
```

¿Porque hago esto? Porque mi proceso puede que este haciendo algo mal, ej: este en un bucle y no sale nunca. Acá el SO lo saca. Le pido al SO que lo termine.

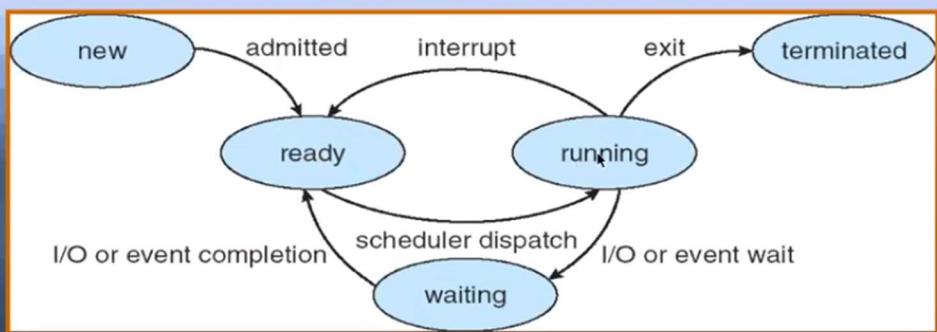
## Jerarquía de Procesos

Relación entre procesos:

- Árbol de procesos
  - Comando pstree
  - Proceso Sched
  - Proceso init o systemd
- Llamadas a sistema:
  - "getpid()"
  - "getppid()"



## Estado de los Procesos



Diferencia entre los estados:

- ¿Cómo cambian de estado?
  - Scheduler .....que??
  - Interrupciones
  - Usuario puede forzar cambio de estado

De runnin a bloqueado pasa mediante una llamada al sistema, de entrada salida.

De bloquea a ready, llega una interrupción, el SO se fija para quien es, y lo pasa a ready.

De ejecución a ready, pasa en sistemas expropiativos.

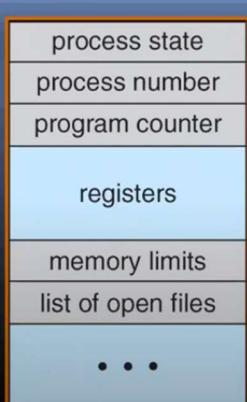
El planificador es una rutina del SO, que elije procesos listos según algún criterio y los pone a ejecutarse.

El tiempo de ejecución depende del operativo y las tareas, pero está en el orden de los 200 mili segundos.

## Bloque de Control Procesos (PCB)

¿Cómo representa el Sistema Operativo los procesos???

- Número de Proceso (PID)
- Estado
- PC ( contador de programa)
- Registros (generales, puntero de pila)
- Espacio de memoria que ocupa
- Descriptores de archivos abiertos
- Usuario / grupo
- etc.

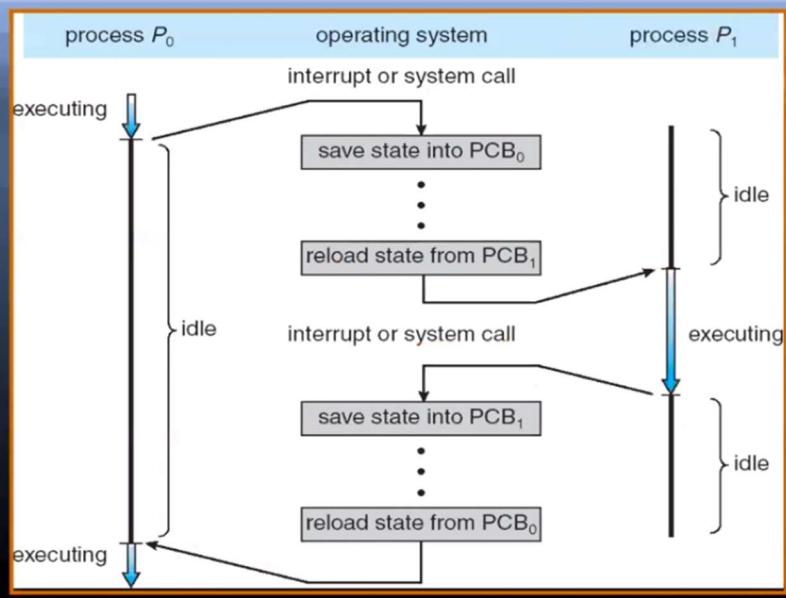


PCB se guardan en la tabla de procesos.

También guarda el PID del padre.

# Comutación entre Procesos (cambio de contexto)

Ejemplo de conmutación de procesos.



Efectos que nos pueden pasar con los procesos. **Procesos Huérfanos y Zombis**. Lo ven en la práctica.

Si un proceso crea un hijo, y el padre termina antes que el hijo. ¿En el PCB que guardo? No puedo almacenar el valor del PID del padre, porque ya terminó. Se lo asocia a otro proceso que siga ejecutándose. Normalmente lo que pasa es que se asocia al proceso 1. (Lo adopta el init o systemd)

```
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ vim fork1.c
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ make fork1
cc      fork1.c -o fork1
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ./fork1
soy el padre pid = 426068 y mi padre es 388229
cree el hijo
soy el hijo pid = 426069 y mi padre es 426068
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ soy el hijo pid = 426069 y mi padre es 1
```

```
int main()
{
    int pid;
    printf("soy el padre pid = %d y mi padre es %d \n", getpid(), getppid());
    pid = fork();
    //hijo
    if (pid == 0){
        printf("soy el hijo pid = %d y mi padre es %d \n", getpid(), getppid());
        sleep(2);
        printf("soy el hijo pid = %d y mi padre es %d \n", getpid(), getppid());
        return 0;
    }
    //    wait(NULL);
    printf("cree el hijo\n");

    return 0;
}
```

Procesos ZOMBIES: Son procesos que terminan, se liberan todos los recursos, no existe en la memoria principal. Solo queda en la tabla de procesos, la entrada a ese proceso en el PCB, porque su padre tiene que ir a leer el "exit status", obviamente el padre todavía está vivo.

Este fenómeno se llama zombie. No ocupa recursos, ni memoria, solo una entrada en la tabla de procesos. Si nosotros programamos mal, se puede dar el caso que tengamos muchos hijos zombies y en algún momento se llene la tabla de procesos, entonces cuando yo quiera crear un nuevo proceso no se pueda.

Desde otra consola. "Con top puedo ver los zombies"

Si termina el padre y el abuelo lee el exit status, el zombie (nieta) desaparece. Salen ambos de la tabla de procesos.

# Señales

## ¿Qué son ?

- Una señal es una notificación asíncrona entregada a un proceso a partir de la ocurrencia de un evento
- Para que se usan ?
  - Aviso del kernel de una excepción de Hardware
    - División por cero, Violación de Segmento, etc.
  - Notificar al proceso de un evento de software
    - Un hijo termina la ejecución, un contador llega a cero, etc.
  - Para controlarlo desde el teclado
    - Suspender el proceso, terminar el proceso.

## Señales:

Son notificaciones asíncronas de software, son muy parecido a una interrupción de hardware.  
Se usan para avisarle, desde el SO al proceso, que sucedió algún evento. En el caso de división de cero, la rutina que atienda esa señal provoca que el proceso termine.

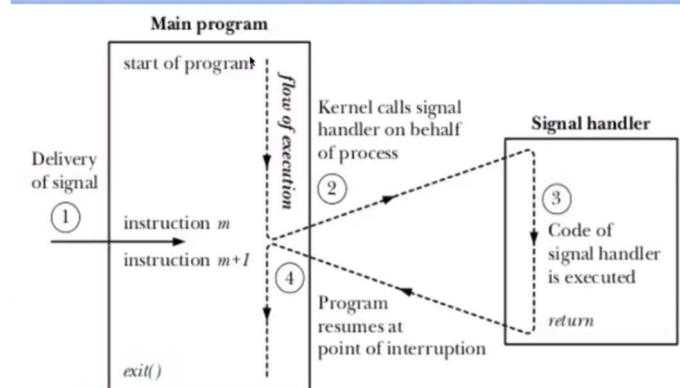
También podría usarlo para mandar estado entre distintos procesos. Supongamos que un proceso trabaja cooperativamente con otro y tiene que leer un resultado que generó el otro proceso. El cual le tiene que avisar cuando ya generó el resultado. Una alternativa es avisarle mediante una señal. Y en una rutina de atención de señal hago que lea de alguna manera el resultado.

“Ctrl + C”: Es una interrupción de hardware, pero cuando la atiende el manejador sabe que debe mandar una señal Sigtint (Signal Interup) al proceso que se está ejecutando, y por defecto termina.

## Manejador de la Señal

### • Cuando un proceso recibe una señal este puede:

- Ignorar la señal.
- Dejar que se ejecute una acción por defecto relacionada con la señal.
- Ejecuta un manejador para la señal: programa encargado de “hacer algo” con la señal recibida.



Para que se ejecuten las rutinas por defecto al llegar una señal no hace falta nada. Si yo quiero ignorarlas o que se haga algo cuando llega esa señal, necesito una biblioteca que realice una llamada al sistema y le avise al SO que cambie el comportamiento, rutina de atención de la señal. (signal)

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

int main (){
    signal(SIGINT, SIG_IGN);
    //luego probar con SIGKILL
//    signal(SIGKILL, SIG_IGN);

    printf("soy el proceso %d\n",getpid());
    printf("estoy trabajando en algo .... para terminar oprima cualquier tecla\n\n");
    int a = getchar();

    printf ("terminando correctamente .... ver exit_status con \"echo $?\n\"");
    return 0;
}
```

Ahora viene un ejemplo, donde llega la señal SIGINT, pero la ignora con sig\_ign

Hay 2 señales que no se pueden ignorar, SigSTOP y Sig KILL. SigStop para detener el proceso hasta nuevo aviso. “Ctrl + z”. Puedo arrancarlo de nuevo con “fg”.

```
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ vim senial1.c
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ make senial
make: *** No rule to make target 'senial'. Stop.
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ make senial1
cc    senial1.c -o senial1
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ./senial1
soy el proceso 426504
estoy trabajando en algo .... para terminar oprima cualquier tecla
^C^Killed
```

Si yo quisiera poner un manejador para sigkill o sigstop, lo voy a poner, pero el programa lo ignora. Tengo que tener una forma de que el proceso termine.

Ósea estas señales no se pueden ignorar ni cambiar su comportamiento.

```

carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ./senial1
soy el proceso 426600
estoy trabajando en algo .... para terminar oprime cualquier tecla
^Z
[1]+ Stopped                  ./senial1
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ fg
./senial1

```

Cuando termina un hijo llega SIGCHILD.  
SIGUSR1 y 2, son para ser utilizadas por el usuario.

Ejemplo de utilizar un manejador, ósea cambiar el comportamiento por defecto de una señal.

```

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void manejador(){
    printf("no pienso terminar\n");
    //signal(SIGINT,SIG_DFL);

}

int main (){
    signal(SIGINT, manejador);

    printf("soy el proceso %d\n",getpid());

    printf("estoy trabajando en algo .... para terminar oprime cualquier tecla\n\n");

    int a = getchar();

    printf ("terminando correctamente .... ver exit_status con \"echo $?\"\n");

    return 0;
}

```

```

carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ./senial2
soy el proceso 426885
estoy trabajando en algo .... para terminar oprime cualquier tecla
^Cno pienso terminar
^Cno pienso terminar
^Cno pienso terminar
no pienso terminar
no pienso terminar
no pienso terminar
^Cno pienso terminar
^Cno pienso terminar
^Cno pienso terminar

```

Ctrl Z lo manda a background, desconecta pantalla y teclado.  
FG: Lo trae a foreground

```

carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ./senial2
soy el proceso 427087
estoy trabajando en algo .... para terminar oprime cualquier tecla
^Z
[1]+ Stopped                  ./senial2
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ./senial2
soy el proceso 427088
estoy trabajando en algo .... para terminar oprime cualquier tecla
^Z
[2]+ Stopped                  ./senial2
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ./senial2
soy el proceso 427091
estoy trabajando en algo .... para terminar oprime cualquier tecla
^Z
[3]+ Stopped                  ./senial2
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ jobs
[1]  Stopped                  ./senial2
[2]- Stopped                  ./senial2
[3]+ Stopped                  ./senial2
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ fg 2
./senial2

```

Jobs: me muestra los procesos bloqueados.

```

void manejador(){
    printf("no pienso terminar\n");
    Signal(SIGINT,SIG_DFL);
}

```

con esa variante, habilita el comportamiento por default de SIGINT, ósea solo ignora el primer ctrl c

```
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ ./senal2
soy el proceso 427240
estoy trabajando en algo .... para terminar oprima cualquier tecla
^Cno pienso terminar
^C
carlost@maquinola:~/tecnicasIII/2021/cap2/Clase2$ echo $?
130
```