Arquitectura del Computador
y
Sistemas Operativos

Vigésimosegunda Clase

Memoria en un ambiente multiproceso

Supongamos que se desea ejecutar dos instancias de un mismo programa a la vez, el SO decide cargar la primera a partir de la posición 0x000 y la segunda a partir de la 0x100:

CARGA A PARTIR DE LA DIRECCIÓN 0x000

```
000: b8 00 00 00 00        mov     eax,0x0
005: a3 80 00 00 00        mov     0x80,eax

0000000a <L0>:
00a: a1 80 00 00 00        mov     eax,0x80
00f: 83 f8 0a              cmp     eax,0xa
012: 74 0d                 je      <L1>
014: ff 05 80 00 00 00     inc     DWORD PTR 0x80
01a: 2e ff 25 0a 00 00 00  jmp     DWORD PTR <L0>

00000021 <L1>:
0021: 90                   nop
```

CARGA A PARTIR DE LA DIRECCIÓN 0x100

```
100: b8 00 00 00 00        mov     eax,0x0
105: a3 80 00 00 00        mov     0x80,eax

0000010a <L0>:
10a: a1 80 00 00 00        mov     eax,0x80
10f: 83 f8 0a              cmp     eax,0xa
112: 74 0d                 je      <L1>
114: ff 05 80 00 00 00     inc     DWORD PTR 0x80
11a: 2e ff 25 0a 00 00 00  jmp     DWORD PTR <L0>

00000121 <L1>:
121: 90                    nop
```

Es evidente que el programa no puede simplemente copiarse a otra dirección y ejecutarse, requiere modificaciones.

Problema 1: Reubicación de ejecutables

Fundamentos

El sistema operativo tiene un módulo destinado a cargar nuevos programas, llamado *loader*. Es el encargado de encontrar en qué lugar de la memoria se ejecutará el programa y de modificarlo para ejecutar en ese lugar.

El proceso de modificar la imágen del programa luego de cargarlo y antes de ejecutarlo se lo conoce como reubicación estática (*static relocation*).

Para poder modificarlo debe saber dónde modificar, cosa que no puede deducir por sí mismo. Ésa información debe estar contenida en el archivo que carga de disco.

El problema de este proceso es que enlentece la carga, dado que hay que sumarle el tiempo de modificar el programa.

Problema 1: Reubicación de ejecutables

Encabezado de un ejecutable DOS

Todo programa DOS comienza con el siguiente encabezado:

DIRECCIÓN	LONGITUD	DESCRIPTION	
0000h	2 bytes	ID='MZ'	
0002h	1 word	Longitud del programa MOD 512	
0004h	1 word	Longitud del programa DIV 512	
0006h	1 word	Número de relocaciones a hacer	
0008h	1 word	Longitud de este header	
000Ah	1 word	Mínima cantidad de párrafos de datos para ejecutar	
000Ch	1 word	Máxima cantidad de párrafos de datos necesarios	
000Eh	1 word	Valor inicial del SS relativa al inicio de la imágen	
0010h	1 word	Valor inicial del SP	
0012h	1 word	Checksum	
0014h	1 dword	CS:IP relativos al inicio de la imágen	
0018h	1 word	Offset a la tabla de relocación	
001Ah	1 word	Número de Overlay (0h = programa principal)	
001Ch	1 dword	Dirección a ajustar relativa al comienzo de la imágen	Tantas como se especifica arriba
xxxxh	xxxx	Imágen del ejecutable	

Ahora el loader no sólo conoce la imágen, sino cómo modificarla para ajustar las direcciones que dependen del lugar donde se la cargue.

Problema 2: Protección entre tareas

Necesidad

Todavía hay un segundo problema, la seguridad.

Cada tarea puede:

- Acceder libremente a los datos de otra
- Modificar los datos de otra tarea sin ser detectada
- Modificar el código de otra tarea. Si el SO mantuviera un listado del usuario que corrió cada proceso para permitirle ciertas cosas, una tarea sin permisos podría modificar el código de otra que sí los tiene para que haga lo que no puede hacer ella misma.

Segmentado (1/6)

Implementación

El *segmentado* consiste dotar al procesador de un registro (o juego de registros) más. Cada vez que tiene una dirección para sacar al bus de direcciones, en vez de salir directamente, se le suma el valor del registro.

Ésto funciona como una reubicación por hardware, y se conoce como *dynamic relocation*. Ya no es necesario modificar el programa, dado que el mismo ejecuta pensando que su dirección de carga es siempre la cero. Ahora el SO sólo tiene que cambiar el valor de ese registro al transferirle el control a cada programa.

Ya que se agrega ese sumador al hardware, resulta útil agregarle además un comparador. Ahora cada segmento tiene una dirección base y un límite. La base se suma a cada dirección que el programa quiere utilizar, y antes de acceder a la misma, se verifica que el valor sea menor que el límite.

El segmentado no solamente resuelve el problema de la reubicación, también resuelve la seguridad.

Solamente necesitamos que el SO sea el único que puede definir y modificar los datos de los segmentos.

Segmentado (2/6)

Procesador 8086

El 8086 tenía registros de segmentación. Si bien era fácil reubicar programas, había dos problemas para la seguridad:

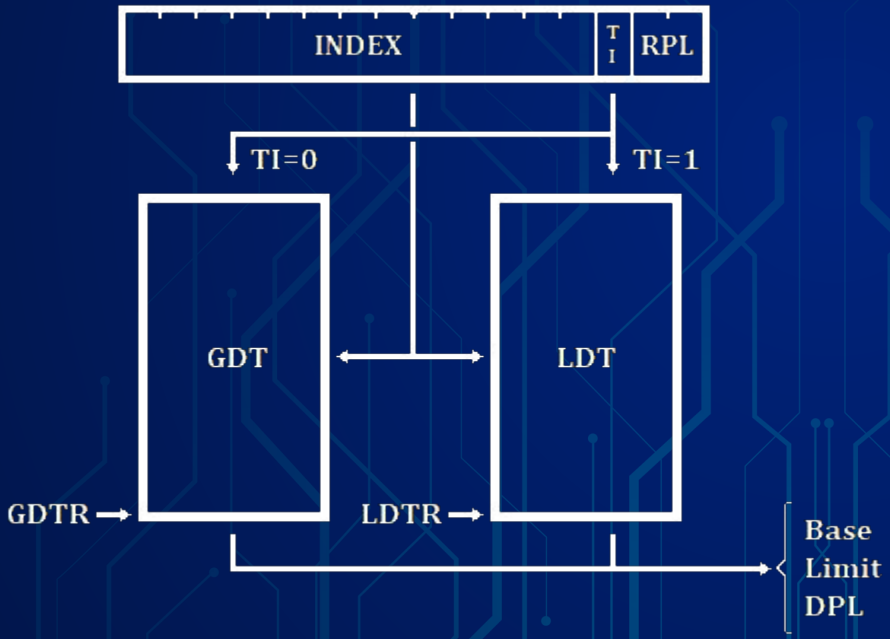
- Los segmentos no tenían un límite.
- Los programas podían modificar los registros de segmentación

Si bien era un procesador de 16 bits (que permitía direccionar 64Kb), los registros de segmentación permitían tener un espacio de direcciones de 20 bits (1Mb)

Segmentado (3/6)

Procesador 80286 (1/2)

El 80286 siguió teniendo instrucciones de 16 bits como su antecesor, el 8086, por lo que cada segmento seguía pudiendo tener un máximo de 64Kb. Sin embargo, extendieron el concepto del segmento.



El 80286 siguió teniendo instrucciones de 16 bits como su antecesor, el 8086, por lo que cada segmento seguía pudiendo tener un máximo de 64Kb. Sin embargo, extendieron el concepto del segmento.

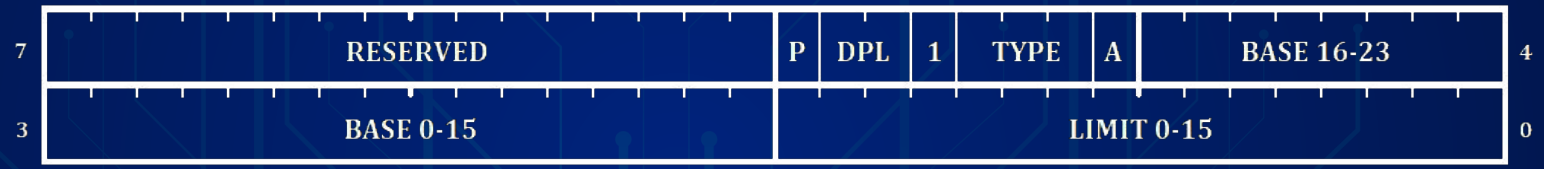
Los registros de segmentación dejaron de tener el valor a sumarle a la dirección lógica para obtener la dirección física, y pasaron a ser punteros a estructuras más complejas.

Estas estructuras ahora tenían la base, el límite y otros valores que permitían implementar la seguridad que faltaba en el 8086.

Segmentado (4/6)

Procesador 80286 (2/2)

Los registros de segmentación tenían partes “ocultas”, donde el procesador cargaba todos los datos que sacaba de la tabla al cargar el segmento. De esta forma no era necesario volver a acceder a la tabla con cada uso.



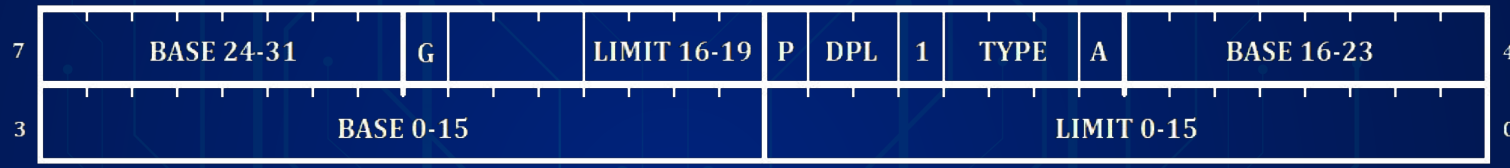
La existencia de dos tablas GDT y LDT permitía tener una tabla para el SO con las funciones expuestas a todos los procesos y una LDT por proceso con todos los segmentos de ese proceso. Así, al cambiar de proceso, sólo había que recargar el LDTR.

Se aprovechó para aumentar la base a 24 bits, ahora el procesador podía tener 16Mb de RAM.

Segmentado (5/6)

Procesador 80386

Se introducen dos cambios. Primero y más importante ahora se incorpora todo el set de instrucciones de 32 bits. Ahora cada segmento podía tener 4Gb.



Además se extendió la base de los segmentos a 32 bits y el límite a 20. Para poder soportar segmentos más grandes de 20 bits, se crea el bit G, que determina si el límite está en bytes o en unidades de 4K

Segmentado (6/6)

La introducción de los registros de segmentación con Base+Límite permitían acotar la memoria a la que podía acceder cada proceso.

Los Privilege Levels vistos en los registros, también se conocen como RINGs. Así, cuando un programa quiere cargar cualquier registro de segmentación, el procesador compara:

- El CPL (o *Current Privilege Level*) que es el privilegio actualmente cargado en el registro de segmentación para el código (CS)
- El RPL (o *Requested Privilege Level*) que es el nivel de privilegio solicitado con el cual usar el segmento a cargar.
- El DPL (o *Descriptor Privilege Level*) que es el máximo nivel que puede usar ese segmento.

Si las verificaciones lo permiten se carga el registro de segmentación, caso contrario se genera una interrupción al SO informando del problema.

De la misma forma, si se trata de usar una posición más allá del límite también se genera una interrupción que es atendida por el SO.

Ahora el SI tiene completo control para aislar a las tareas entre sí.

Memoria Swapping (1/2)

En cualquiera de los escenarios vistos, si se desea ejecutar una nueva tarea y no hay memoria, el SO puede copiar la imagen en memoria de algún proceso y liberar el lugar necesario para correrla.

Cuando no había apoyo del hardware, el SO debía elegir a arbitrariamente una tarea para bajar. Una vez elegida quedaba completamente inactiva hasta que se decidiera volver a subirla a RAM. Ésto era así aunque la tarea tuviera un gran bloque de memoria que no estaba en uso y en el cual entraría el proceso nuevo que se quiere correr. Ésto se conoce como *full-swap*.

En momentos de la historia en que la necesidad de memoria en los programas creció más rápidamente que la disponible en los ordenadores, los programadores recurrieron el concepto de *Overlays*. Esta técnico consistía en dividir el código en bloques elegidos con detalla de forma tal que no todo el programa tuviera que estar en memoria a la vez.

Lo que se veía del lado del usuario, por ejemplo al ingresar al autocorrector de un procesador de texto, era una pequeña demora que coincidía con la luz de acceso a disco. Lo que estaba pasando es que el SO estaba bajando a disco, a pedido del mismo programa digamos el módulo de definición de gráficos para subir el del autocorrector.

Memoria Swapping (2/2)

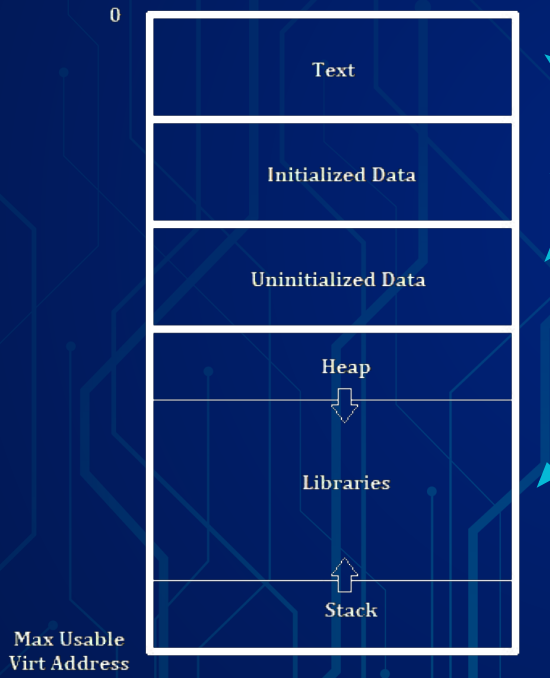
Pero con el apoyo de hardware del sistema de segmentado hay una nueva opción:

- El bit A de la tabla de segmentos es subido a 1 cada vez que el segmento se usa. De esta forma, borrando este bit periódicamente el SO puede determinar qué segmentos no se están usando.
- Al detectar uno sin uso, puede bajar ese segmento a disco y colocarle el bit P (present) en 0. Cualquier intento de uso del segmento en esas condiciones resultará en un trap al SO.
- En el lugar liberado, colocar la nueva tarea

La nueva tarea está corriendo *sin bajar ninguna otra*. Y sólo si la tarea a la que se le sacó el segmento intentara usarlo habría que detenerla.

Estructura de un Programa C / C++ (1/3)

Cuando se compila un programa en C/C++, el código generado y las variables y/o estructuras de datos generadas se colocan en distintas zonas de memoria (o segmentos). Una distribución típica es la siguiente:




```
int i = 0;
int j;

int main(int argc, char *argv[])
{
j=2*i;
printf("El resultado es %d\n", j);
return(0);
}
```

Estructura de un Programa C / C++ (2/3)

Algunos bloques, como el de código, son estáticos. No pueden crecer una vez cargados, pero el caso de los bloques de datos es distinto.

Con el sistema de segmentado, si se da una situación como la de la figura y el Segmento 1 necesita crecer, el SO puede:

- Esperar que la tarea que usa el Segmento 2 no esté ejecutando
- Mover el contenido del Segmento 2 hacia abajo
- Ajustar la base del Segmento 2.
- Aumentar el límite del Segmento 1

Si bien la tarea es costosa en términos de tiempo (mover todo el contenido del segmento hacia abajo lleva tiempo), las tareas pueden ir pidiendo memoria y el SO entregándola sin que haya desperdicios.

Estructura de un Programa C / C++ (3/3)

Podemos utilizar el programa *size* para ver qué segmentos define un programa:

```
[root@server bin]# size ./progname
   text    data     bss     dec     hex filename
 921304   32328  907784 1861416  1c6728 ./progname
```

Si necesitamos información más detallada, podemos usar el comando *readelf*

```
[root@server bin]# readelf -l ./progname

Elf file type is EXEC (Executable file)
Entry point 0x43e210
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr           FileSiz            MemSiz             Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040 0x00000000000001f8 0x00000000000001f8  R E    0x8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238 0x000000000000001c 0x000000000000001c  R      0x1
                 [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000 0x00000000000e112c 0x00000000000e112c  R E    0x200000
  LOAD           0x00000000000e1c88 0x00000000006e1c88 0x00000000006e1c88 0x0000000000007e58 0x00000000000e5860  RW     0x200000
  DYNAMIC        0x00000000000e1dc0 0x00000000006e1dc0 0x00000000006e1dc0 0x0000000000000230 0x0000000000000230  RW     0x8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x00000000000c73f0 0x00000000004c73f0 0x00000000004c73f0 0x000000000000450c 0x000000000000450c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x00000000000e1c88 0x00000000006e1c88 0x00000000006e1c88 0x0000000000000378 0x0000000000000378  R      0x1
```

Fin

¿Preguntas?