

Práctica de Organización del Computador II

Tareas

Segundo Cuatrimestre 2023

Organización del Computador II
DC - UBA

Introducción

En la clase de hoy vamos a ver:

En la clase de hoy vamos a ver:

- **Repaso de tareas y scheduler**

En la clase de hoy vamos a ver:

- Repaso de tareas y scheduler
- Estructuras involucradas en tareas (TR, TSS Descriptor en GDT, TSS)

En la clase de hoy vamos a ver:

- Repaso de tareas y scheduler
- Estructuras involucradas en tareas (TR, TSS Descriptor en GDT, TSS)
- Atributos relevantes de las estructuras

En la clase de hoy vamos a ver:

- Repaso de tareas y scheduler
- Estructuras involucradas en tareas (TR, TSS Descriptor en GDT, TSS)
- Atributos relevantes de las estructuras
- Cómo se produce un cambio de tarea

En la clase de hoy vamos a ver:

- **Repaso de tareas y scheduler**
- **Estructuras involucradas en tareas (TR, TSS Descriptor en GDT, TSS)**
- **Atributos relevantes de las estructuras**
- **Cómo se produce un cambio de tarea**

La idea es presentar la información necesaria para ayudarles a completar el taller.

Repaso de tareas

Las computadoras ejecutan varios programas en simultáneo a la vista de los usuarios. Por ejemplo, mientras navegamos por la web podemos utilizar una aplicación para escuchar música.

Sin embargo, como vimos, cada procesador ejecuta un programa por vez. ¿Cómo se logra esto el sistema operativo?

Las computadoras ejecutan varios programas en simultáneo a la vista de los usuarios. Por ejemplo, mientras navegamos por la web podemos utilizar una aplicación para escuchar música.

Sin embargo, como vimos, cada procesador ejecuta un programa por vez. ¿Cómo se logra esto el sistema operativo?

Para comprenderlo, vamos a usar una serie de estructuras y funciones de Intel que nos permiten definir tareas para el procesador.

Las computadoras ejecutan varios programas en simultáneo a la vista de los usuarios. Por ejemplo, mientras navegamos por la web podemos utilizar una aplicación para escuchar música.

Sin embargo, como vimos, cada procesador ejecuta un programa por vez. ¿Cómo se logra esto el sistema operativo?

Para comprenderlo, vamos a usar una serie de estructuras y funciones de Intel que nos permiten definir tareas para el procesador.

A su vez, el sistema operativo va a implementar un módulo de software que se va a encargar de decidir que tarea ejecutar en cada tic del reloj: scheduler.

Una tarea es una unidad de trabajo que el procesador puede despachar, ejecutar, y suspender. Puede ser usada para ejecutar un programa.

Una tarea es una unidad de trabajo que el procesador puede despachar, ejecutar, y suspender. Puede ser usada para ejecutar un programa.

Dos o más tareas distintas pueden tener un mismo código de programa, sin embargo, sus contexto de ejecución y datos asociados pueden ser distintos. Podemos pensarlo como distintas instancias del mismo programa.

En memoria una tarea va a tener:

En memoria una tarea va a tener:

1. **Espacio de Ejecución:** Es decir, páginas mapeadas donde va a tener el código, datos y pilas. Podríamos precisar definirle un page directory con su correspondiente pages table o reutilizar algún directorio entre varias tareas.

En memoria una tarea va a tener:

1. **Espacio de Ejecución:** Es decir, páginas mapeadas donde va a tener el código, datos y pilas. Podríamos precisar definirle un page directory con su correspondiente pages table o reutilizar algún directorio entre varias tareas.
2. **Segmento de Estado (TSS):** Una región de memoria que almacena el estado de una tarea, a la espera de iniciarse o al momento de ser desalojada del procesador, y con un formato específico para que podamos iniciarla/reanudarla. La información que se va a guardar en esta región sería:

En memoria una tarea va a tener:

1. **Espacio de Ejecución:** Es decir, páginas mapeadas donde va a tener el código, datos y pilas. Podríamos precisar definirle un page directory con su correspondiente pages table o reutilizar algún directorio entre varias tareas.
2. **Segmento de Estado (TSS):** Una región de memoria que almacena el estado de una tarea, a la espera de iniciarse o al momento de ser desalojada del procesador, y con un formato específico para que podamos iniciarla/reanudarla. La información que se va a guardar en esta región sería:
 - Registros de propósito general
 - Registros de segmento de la tarea y segmento de la pila de nivel 0
 - Flags
 - CR3
 - EIP

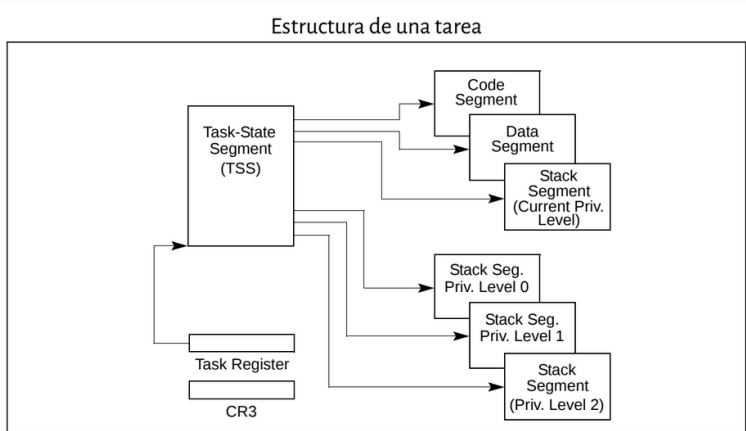
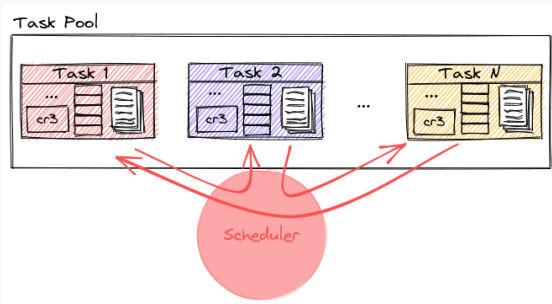


Figure 7-1. Structure of a Task

El **scheduler** es un módulo de software que administra la ejecución de tareas / procesos. Utiliza una política o criterio para decir cuál es la próxima tarea a ejecutar.

El **scheduler** es un módulo de software que administra la ejecución de tareas / procesos. Utiliza una política o criterio para decir cuál es la próxima tarea a ejecutar.



Cada vez que se pasa de una tarea a otra ocurre un **Cambio de Contexto**

Como mencionamos, el procesador puede correr una única tarea por vez y cada tarea tiene su propio contexto de ejecución.

Como mencionamos, el procesador puede correr una única tarea por vez y cada tarea tiene su propio contexto de ejecución.

Al pasar de una a otra, tiene que ir cambiando el contexto de manera acorde.

Como mencionamos, el procesador puede correr una única tarea por vez y cada tarea tiene su propio contexto de ejecución.

Al pasar de una a otra, tiene que ir cambiando el contexto de manera acorde.

Esto significa que tiene que guardar en algún lado el contexto actual de la tarea para no perderlo si la tiene que continuar ejecutando y cargar el contexto de la nueva tarea a ejecutar.

Como mencionamos, el procesador puede correr una única tarea por vez y cada tarea tiene su propio contexto de ejecución.

Al pasar de una a otra, tiene que ir cambiando el contexto de manera acorde.

Esto significa que tiene que guardar en algún lado el contexto actual de la tarea para no perderlo si la tiene que continuar ejecutando y cargar el contexto de la nueva tarea a ejecutar.

El procesador se encarga de ir copiando esta información en cada cambio de contexto.

Estructuras involucradas

Para definir una tarea, tenemos que completar estructuras de Intel:

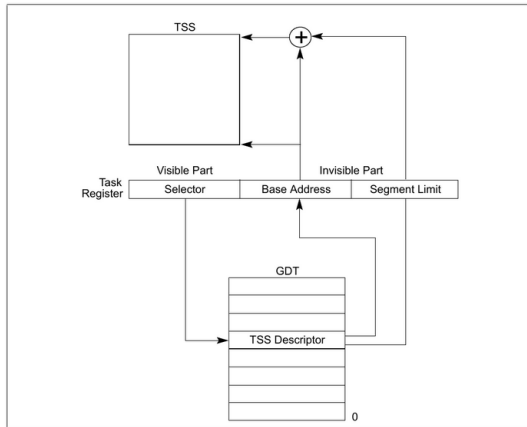
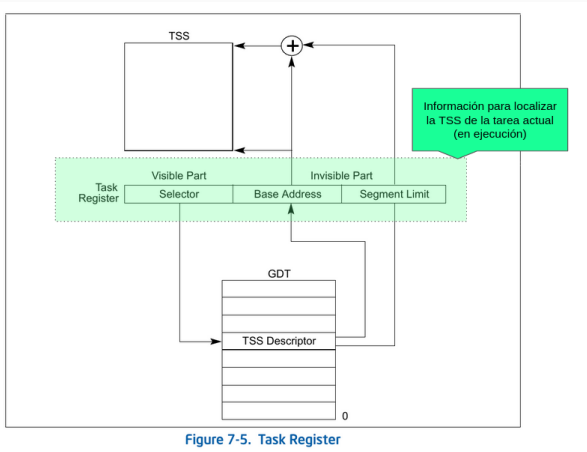
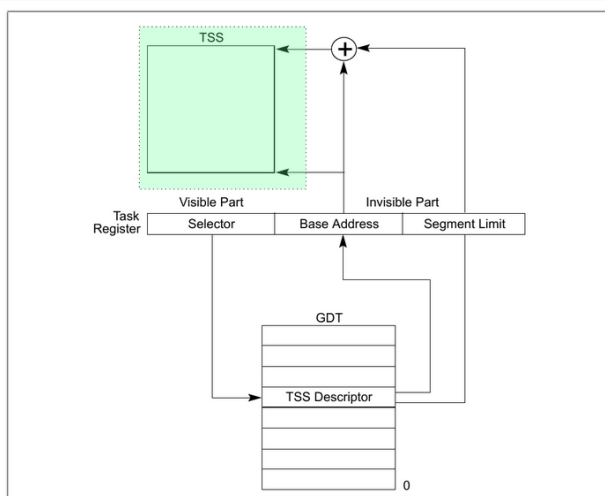


Figure 7-5. Task Register

El registro Task Register almacena el selector de segmento de la tarea en ejecución.

Se usa para encontrar la TSS de la tarea actual.





Segmento de Estado de Tarea - TSS

31	15	0	
I/O Map Base Address	Reserved	T	100
Reserved	LDT Segment Selector		96
Reserved	GS		92
Reserved	FS		88
Reserved	DS		84
Reserved	SS		80
Reserved	CS		76
Reserved	ES		72
	EDI		68
	ESI		64
	EBP		60
	ESP		56
	EBX		52
	EDX		48
	ECX		44
	EAX		40
	EFLAGS		36
	EIP		32
	CR3 (PDBR)		28
Reserved	SS2		24
	ESP2		20
Reserved	SS1		16
	ESP1		12
Reserved	SS0		8
	ESP0		4
Reserved	Previous Task Link		0

La TSS guarda una foto del contexto de ejecución de la tarea.

Al crear la tarea, hay que setear los valores iniciales.

Para inicializar la TSS de una tarea, tenemos que completar con la información inicial que posibilite la correcta ejecución de la tarea. Es decir, los valores que va a tener son aquellos que se van a cargar en los registros de CPU y que usará en la ejecución. Los siguientes son los campos más relevantes a completar:

Para inicializar la TSS de una tarea, tenemos que completar con la información inicial que posibilite la correcta ejecución de la tarea. Es decir, los valores que va a tener son aquellos que se van a cargar en los registros de CPU y que usará en la ejecución. Los siguientes son los campos más relevantes a completar:

- **EIP**

Para inicializar la TSS de una tarea, tenemos que completar con la información inicial que posibilite la correcta ejecución de la tarea. Es decir, los valores que va a tener son aquellos que se van a cargar en los registros de CPU y que usará en la ejecución. Los siguientes son los campos más relevantes a completar:

- **EIP**
- **ESP, EBP y ESP0** (puntero al tope de pila de nivel 0)

Para inicializar la TSS de una tarea, tenemos que completar con la información inicial que posibilite la correcta ejecución de la tarea. Es decir, los valores que va a tener son aquellos que se van a cargar en los registros de CPU y que usará en la ejecución. Los siguientes son los campos más relevantes a completar:

- **EIP**
- **ESP, EBP y ESP0** (puntero al tope de pila de nivel 0)
- Los selectores de segmento **CS, DS, ES, FS, GS, SS, SS0** (selector de la pila de nivel 0)

Para inicializar la TSS de una tarea, tenemos que completar con la información inicial que posibilite la correcta ejecución de la tarea. Es decir, los valores que va a tener son aquellos que se van a cargar en los registros de CPU y que usará en la ejecución. Los siguientes son los campos más relevantes a completar:

- **EIP**
- **ESP, EBP y ESP0** (puntero al tope de pila de nivel 0)
- Los selectores de segmento **CS, DS, ES, FS, GS, SS, SS0** (selector de la pila de nivel 0)
- El **CR3** que va a tener la paginación asociada a la tarea. Cada tarea tendrá así su propio directorio de páginas.

Para inicializar la TSS de una tarea, tenemos que completar con la información inicial que posibilite la correcta ejecución de la tarea. Es decir, los valores que va a tener son aquellos que se van a cargar en los registros de CPU y que usará en la ejecución. Los siguientes son los campos más relevantes a completar:

- **EIP**
- **ESP, EBP y ESP0** (puntero al tope de pila de nivel 0)
- Los selectores de segmento **CS, DS, ES, FS, GS, SS, SS0** (selector de la pila de nivel 0)
- El **CR3** que va a tener la paginación asociada a la tarea. Cada tarea tendrá así su propio directorio de páginas.
- **EFLAGS** en 0x00000202 para tener las interrupciones habilitadas

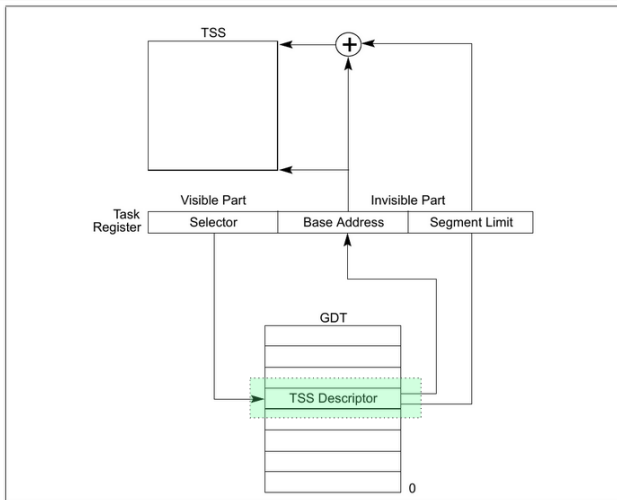
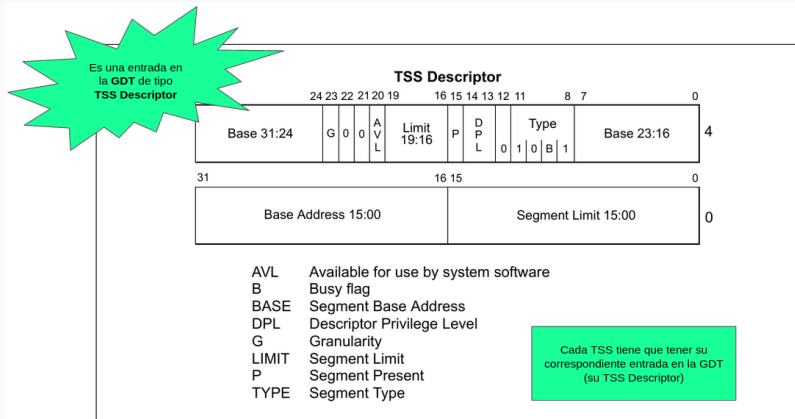


Figure 7-5. Task Register



- El bit **B**(Busy) indica si la tarea está siendo ejecutada. Lo iniciamos en 0.

- El bit **B**(Busy) indica si la tarea está siendo ejecutada. Lo iniciamos en 0.
- El bit **DPL**(Descriptor Privilege Level) el nivel de privilegio que se precisa para acceder al segmento. Usamos nivel 0 porque sólo el kernel puede intercambiar tareas.

- El bit **B**(Busy) indica si la tarea está siendo ejecutada. Lo iniciamos en 0.
- El bit **DPL**(Descriptor Privilege Level) el nivel de privilegio que se precisa para acceder al segmento. Usamos nivel 0 porque sólo el kernel puede intercambiar tareas.
- El **LIMIT** es el tamaño máximo de la TSS. 67h es el mínimo requerido.

- El bit **B**(Busy) indica si la tarea está siendo ejecutada. Lo iniciamos en 0.
- El bit **DPL**(Descriptor Privilege Level) el nivel de privilegio que se precisa para acceder al segmento. Usamos nivel 0 porque sólo el kernel puede intercambiar tareas.
- El **LIMIT** es el tamaño máximo de la TSS. 67h es el mínimo requerido.
- El **BASE** indica la dirección base de la TSS

Definición de tareas Inicial e Idle

El procesador siempre precisa estar ejecutando una tarea, aunque esta no haga nada. Hay dos situaciones especiales:

El procesador siempre precisa estar ejecutando una tarea, aunque esta no haga nada. Hay dos situaciones especiales:

- Al arrancar la computadora, ¿qué tarea se ejecuta?

El procesador siempre precisa estar ejecutando una tarea, aunque esta no haga nada. Hay dos situaciones especiales:

- Al arrancar la computadora, ¿qué tarea se ejecuta?
- Y si no hubiera tarea para ejecutar en algún momento, ¿qué tarea se ejecuta?

El procesador siempre precisa estar ejecutando una tarea, aunque esta no haga nada. Hay dos situaciones especiales:

- Al arrancar la computadora, ¿qué tarea se ejecuta?
- Y si no hubiera tarea para ejecutar en algún momento, ¿qué tarea se ejecuta?

Necesitamos definir dos tareas especiales: la **tarea Inicial** y la **tarea Idle** para estas situaciones. Además, definiremos aquellas tareas de usuario y/o de kernel que se precisan para que nuestro sistema brinde servicios o haga lo que esperamos.

Necesitamos dos pasos para dejar al kernel listo para ejecutar las tareas que querramos:

Necesitamos dos pasos para dejar al kernel listo para ejecutar las tareas que querramos:

1. Apenas inicia el kernel hay que cargar la **tarea Inicial**. Para hacerlo, vamos a usar la instrucción LTR que toma como parámetro un registro de 16 bits con el selector de la tarea en la GDT.

LDTR ax ; (con ax = selector segmento tarea inicial)

Necesitamos dos pasos para dejar al kernel listo para ejecutar las tareas que querramos:

1. Apenas inicia el kernel hay que cargar la **tarea Inicial**. Para hacerlo, vamos a usar la instrucción LTR que toma como parámetro un registro de 16 bits con el selector de la tarea en la GDT.

LDTR ax ; (con ax = selector segmento tarea inicial)

2. Luego, hay que saltar a la **tarea Idle**. La forma de hacerlo es saltar al selector con un JMP y el valor que pongamos en offset es ignorado (podemos poner 0).

JMP SELECTOR_TAREA_IDLE:0

Necesitamos dos pasos para dejar al kernel listo para ejecutar las tareas que querramos:

1. Apenas inicia el kernel hay que cargar la **tarea Inicial**. Para hacerlo, vamos a usar la instrucción LTR que toma como parámetro un registro de 16 bits con el selector de la tarea en la GDT.

LDTR ax ; (con ax = selector segmento tarea inicial)

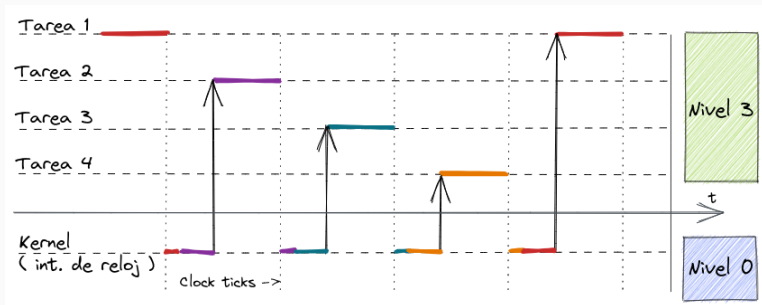
2. Luego, hay que saltar a la **tarea Idle**. La forma de hacerlo es saltar al selector con un JMP y el valor que pongamos en offset es ignorado (podemos poner 0).

JMP SELECTOR_TAREA_IDLE:0

Esto va a cambiar el valor del registro TR apuntando a la TSS de la tarea Idle y producir el cambio de contexto. Saltar a una tarea es algo que lo va a hacer el Sistema Operativo en nivel 0.

Intercambio de Tareas

Utiliza una política o criterio para decir cuál es la próxima tarea a ejecutar y lo hace en cada tic del reloj

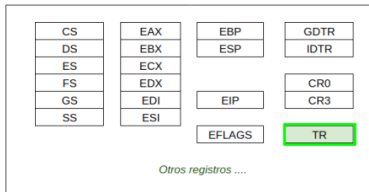


Cada vez que se pasa de una tarea a otra ocurre un **Cambio de Contexto**

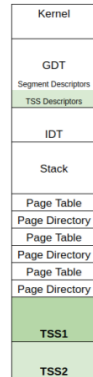
El procesador se encuentra siempre ejecutando una tarea

La tarea actual está indicada por el registro TR

CPU



Memoria

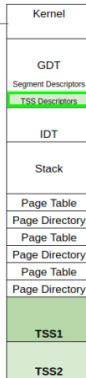


En un tic del reloj, el scheduler le pide cambiar la tarea en ejecución haciendo `jmp` al segmento de la TSS de la nueva tarea a ejecutar en la GDT

CPU



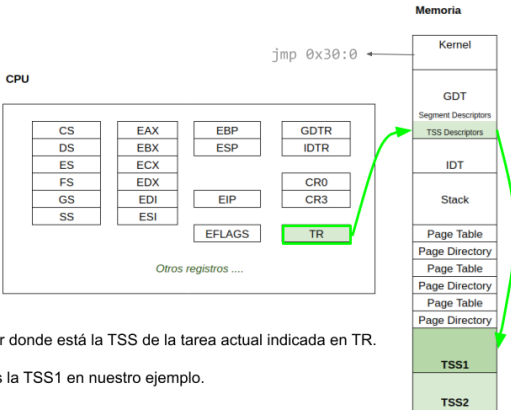
Memoria



`jmp 0x30:0`

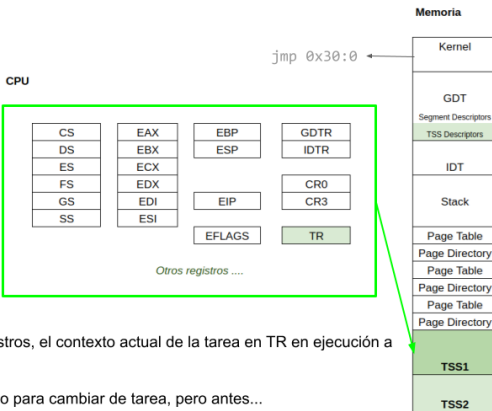
Debe realizar un intercambio de contexto (Context Switch).

Antes de cambiar la tarea, copia el estado actual de los registros, sus valores, en la TSS de la tarea en ejecución indicada por TR.



Para eso, primero, precisa conocer donde está la TSS de la tarea actual indicada en TR.

Utiliza la GDT para encontrarla. Es la TSS1 en nuestro ejemplo.



Copia los registros, el contexto actual de la tarea en TR en ejecución a su TSS.

Ahora está listo para cambiar de tarea, pero antes...

Ahora está listo para cambiar de tarea, pero antes...

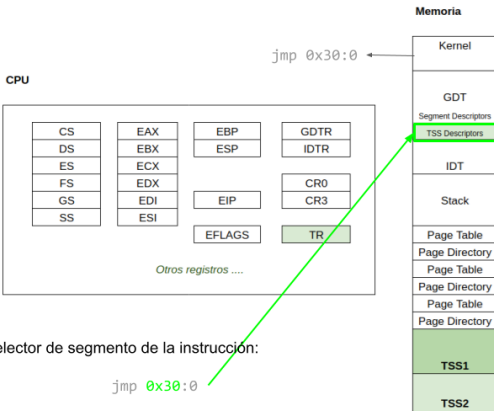
Debe resumir la nueva tarea, es decir, continuar su ejecución de donde la había dejado anteriormente.

Es decir, copia aquellos valores que guardó desde su TSS hacia los registros.

Con el selector de segmento de la instrucción:

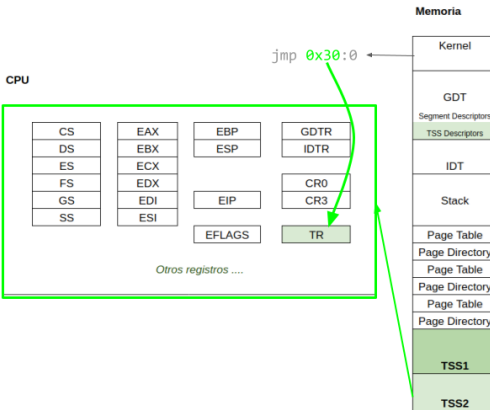
`jmp 0x30:0`

usa la GDT para encontrar la TSS de la nueva tarea



Copia los registros de la nueva tarea desde su TSS en memoria hacia la CPU

Actualiza el TR con la nueva tarea para indicar la tarea actual en ejecución

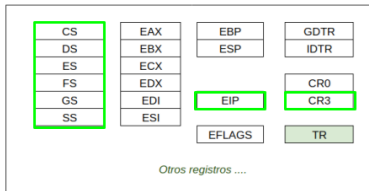


Noten que además de modificar los registros de propósito general, modifica los de segmentos, el EIP y el CR3.

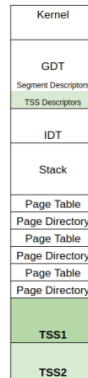
Con lo cual, va a cambiar la paginación y la dirección de la próxima instrucción a ejecutar (EIP).

Va a ejecutar otro programa.

CPU



Memoria



```
offset: dd 0  
selector: dw 0  
...
```

```
global _isr32
```

```
_isr32:  
    pushad  
  
    call pic_finish1  
    call sched_nextTask  
  
    str cx  
    cmp ax, cx  
    je .fin
```

```
    mov [selector], ax  
    jmp far [offset]
```

```
.fin:  
    popad  
    iret
```

En algún lugar, se definen offset y selector. La estructura definida se puede ver como una dirección lógica de 48 bits en little endian

Y la siguiente, es una rutina de atención de interrupción del reloj (#32)



```
offset: dd 0  
selector: dw 0
```

```
...
```

```
global _isr32
```

```
_isr32:
```

```
    pushad
```

```
    call pic_finish1
```

```
    call sched_nextTask
```

```
    str cx
```

```
    cmp ax, cx
```

```
    je .fin
```

```
    mov [selector], ax
```

```
    jmp far [offset]
```

```
.fin:
```

```
    popad
```

```
    iret
```

Veamos que hace la Rutina de Atención de Interrupción del reloj dada:

Primero, hace pushad y su correspondiente popad para guardar/obtener los registros de propósito general.



```
offset: dd 0
selector: dw 0
...

global _isr32

_isr32:
    pushad

    call pic_finish1
    call sched_nextTask

    str cx
    cmp ax, cx
    je .fin

    mov [selector], ax
    jmp far [offset]

.fin:
    popad
    iret
```

← iret para volver a la rutina que la llamó resturando el EIP



```
offset: dd 0
selector: dw 0
...

global _isr32

_isr32:
    pushad

    call pic_finish1
    call sched_nextTask

    str cx
    cmp ax, cx
    je .fin

    mov [selector], ax
    jmp far [offset]

.fin:
    popad
    iret
```

← indica al PIC que la interrupción fue atendida

```
offset: dd 0
selector: dw 0
...

global _isr32

_isr32:
    pushad

    call pic_finish1
    call sched_nextTask

    str cx
    cmp ax, cx
    je .fin

    mov [selector], ax
    jmp far [offset]

.fin:
    popad
    iret
```

← Intercambio de tareas!!



```
call sched_nextTask
```

```
str cx  
cmp ax, cx  
je .fin
```

```
mov [selector], ax  
jmp far [offset]
```

Pide al scheduler la próxima tarea a ejecutar.

Devuelve la próxima tarea con un valor guardado en ax,

¿qué debería ser ese valor?

¿por qué usa ax y no eax si estamos en 32 bits?



```
call sched_nextTask
```

```
str cx  
cmp ax, cx  
je .fin
```

```
mov [selector], ax  
jmp far [offset]
```

Pide al scheduler la próxima tarea a ejecutar.

Devuelve la próxima tarea con un valor guardado en ax,

¿qué debería ser ese valor?

¿por qué usa ax y no eax si estamos en 32 bits?

El método `sched_nextTask` del scheduler devuelve en ax el selector de segmento de la próxima tarea a ejecutar.

Los selectores tiene 16 bits por eso usa ax y no eax.



```
call sched_nextTask
```

```
str cx  
cmp ax, cx  
je .fin
```

```
mov [selector], ax  
jmp far [offset]
```

STR lee el registro TR y lo guarda en el registro de propósito general usado como operador.

Es decir, en este caso, **guarda en cx el valor de TR.**

Ahora, cx va a tener el valor del selector del segmento de la tarea en ejecución

```
call sched_nextTask
```

```
str cx  
cmp ax, cx  
je .fin
```

```
mov [selector], ax  
jmp far [offset]
```

ax ← selector de segmento de la **tarea próxima**

cx ← selector de segmento de la **tarea actual** (en ejecución)

¿Qué hacen `cmp ax, cx` y `je .fin`?

```
call sched_nextTask
```

```
str cx  
cmp ax, cx  
je .fin
```

```
mov [selector], ax  
jmp far [offset]
```

ax ← selector de segmento de la **tarea próxima**

cx ← selector de segmento de la **tarea actual** (en ejecución)

¿Qué hacen `cmp ax, cx` y `je .fin`?

Si la tarea en ejecución es la misma que la próxima (`ax = cx`), salta a fin y no hay cambio de tarea.

Si son distintas.... ejecutas las siguientes líneas

```
call sched_nextTask
```

```
str cx  
cmp ax, cx  
je .fin
```

```
mov [selector], ax  
jmp far [offset]
```

Si son distintas.... ejecutas las siguientes líneas..

Donde mueve el valor de ax a la posición de memoria reservada para el selector.

Y luego, hace un jmp far al contenido de la dirección indicada por offset

Dicho jump recibe una dirección lógica de 48 bits... y habíamos definido al comienzo...

```
offset: dd 0  
selector: dw 0
```

y ahora

```
offset: dd 0  
selector: tiene el valor de ax
```

¿Qué significa esto?



```
call sched_nextTask
```

```
str cx  
cmp ax, cx  
je .fin
```

```
mov [selector], ax  
jmp far [offset]
```

y ahora

offset: dd 0

selector: tiene el valor de ax

¿Qué significa esto?

Significa que **termina saltando al selector de TSS en la GDT de la tarea próxima retornada por el scheduler.**

Cambia la tarea y automáticamente se **dispara el cambio de contexto.**

```
offset: dd 0
selector: dw 0
...
```

```
global _isr32
```

```
_isr32:
```

```
  pushad
```

```
  call pic_finish1
```

```
  call sched_nextTask
```

```
  str cx
```

```
  cmp ax, cx
```

```
  je .fin
```

```
  mov [selector], ax
```

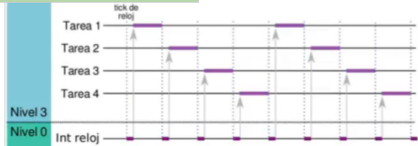
```
  jmp far [offset]
```

```
.fin:
```

```
  popad
```

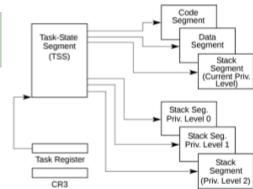
```
  iret
```

Cada tic del reloj se ejecuta esta rutina de atención de interrupción



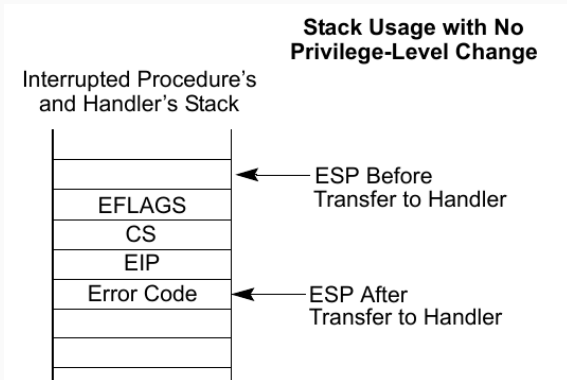
Y cada cambio de tarea que haga va a producir un cambio de contexto

CPU

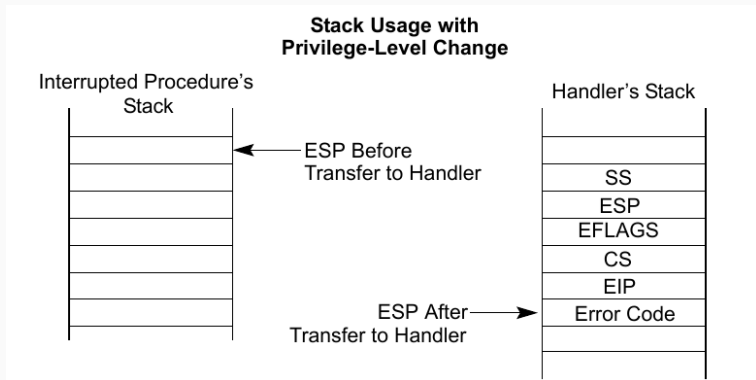


Estructura de una tarea

Imaginemos una tarea ejecutando en nivel 0 indicado por su ss y se produce la interrupción de reloj. El nivel de ejecución no cambia dado que la interrupción de reloj es nivel 0.

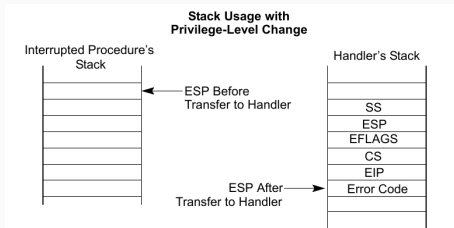


Ahora, si tenemos una tarea ejecutando en nivel 3 indicado por su `ss` y se produce la interrupción de reloj. El nivel de ejecución cambia. Por lo tanto, usa la pila de nivel 0 (`ss0`) indicada en la TSS para guardar la información de retorno.



Cuando hay niveles de privilegios distintos, la ss y esp del procesador siempre toma la del nivel de ejecución actual. Ejecutando una tarea de nivel 3 y justo se produjo una interrupción de nivel 0. Si se produce un cambio de contexto, la TSS de una tarea de nivel 3 podría quedar con un ss almacenado de nivel 0. Los valores nivel 3 quedan en la pila y se restaurarán en el iret correspondiente.

31	15	0	
I/O Map Base Address	Reserved		100
Reserved	LDT Segment Selector		96
Reserved	GS		92
Reserved	FS		88
Reserved	DS		84
Reserved	SS		80
Reserved	CS		76
Reserved	ES		72
	EDI		68
	ESI		64
	EBP		60
	ESP		56
	EBX		52
	EDX		48
	ECX		44
	EAX		40
	EFLAGS		36
	EIP		32
	CR3 (PDBR)		28
Reserved	SS2		24
Reserved	ESP2		20
Reserved	SS1		16
Reserved	ESP1		12
Reserved	SS0		8
Reserved	ESP0		4
Reserved	Previous Task Link		0



En la introducción de hoy vimos:

- Que el procesador ejecuta los programas en forma de tareas

En la introducción de hoy vimos:

- Que el procesador ejecuta los programas en forma de tareas
- Que estas tienen un **TSS descriptor** en la GDT que indica la ubicación de su TSS en memoria

En la introducción de hoy vimos:

- Que el procesador ejecuta los programas en forma de tareas
- Que estas tienen un **TSS descriptor** en la GDT que indica la ubicación de su TSS en memoria
- Que la próxima tarea a ejecutar lo decide un módulo llamado **scheduler** en cada tic de reloj

En la introducción de hoy vimos:

- Que el procesador ejecuta los programas en forma de tareas
- Que estas tienen un **TSS descriptor** en la GDT que indica la ubicación de su TSS en memoria
- Que la próxima tarea a ejecutar lo decide un módulo llamado **scheduler** en cada tic de reloj
- Cómo se guarda el contexto de cada tarea desalojada del procesador en **TSS**

En la introducción de hoy vimos:

- Que el procesador ejecuta los programas en forma de tareas
- Que estas tienen un **TSS descriptor** en la GDT que indica la ubicación de su TSS en memoria
- Que la próxima tarea a ejecutar lo decide un módulo llamado **scheduler** en cada tic de reloj
- Cómo se guarda el contexto de cada tarea desalojada del procesador en **TSS**
- El registro que indica la tarea actual en ejecución: **TR**

En la introducción de hoy vimos:

- Que el procesador ejecuta los programas en forma de tareas
- Que estas tienen un **TSS descriptor** en la GDT que indica la ubicación de su TSS en memoria
- Que la próxima tarea a ejecutar lo decide un módulo llamado **scheduler** en cada tic de reloj
- Cómo se guarda el contexto de cada tarea desalojada del procesador en **TSS**
- El registro que indica la tarea actual en ejecución: **TR**
- Cómo se produce el **cambio de contexto**

En la introducción de hoy vimos:

- Que el procesador ejecuta los programas en forma de tareas
- Que estas tienen un **TSS descriptor** en la GDT que indica la ubicación de su TSS en memoria
- Que la próxima tarea a ejecutar lo decide un módulo llamado **scheduler** en cada tic de reloj
- Cómo se guarda el contexto de cada tarea desalojada del procesador en **TSS**
- El registro que indica la tarea actual en ejecución: **TR**
- Cómo se produce el **cambio de contexto**
- La necesidad de definir una **tarea Inicial** y una **tarea Idle** además de las tareas de usuario

Consultas y ejercitación
