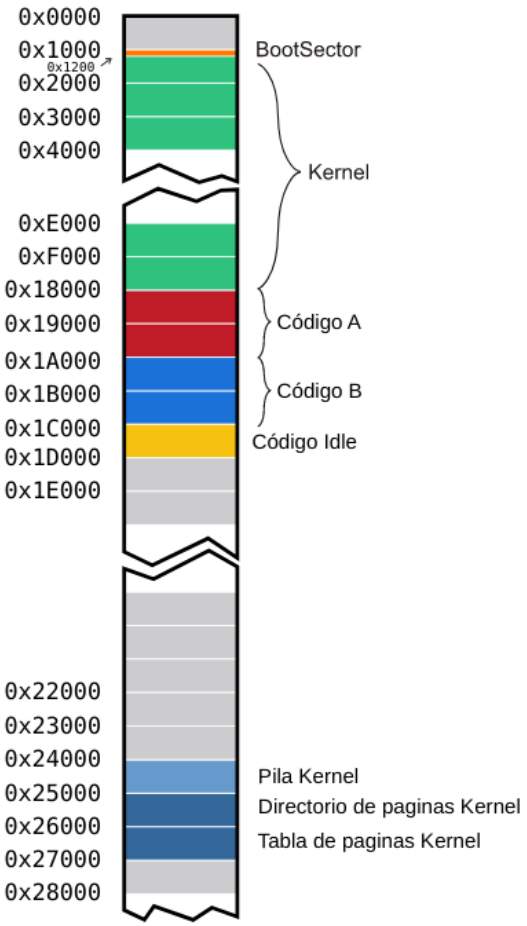
Organización del Computador II
System Programming
Taller 4: Tareas

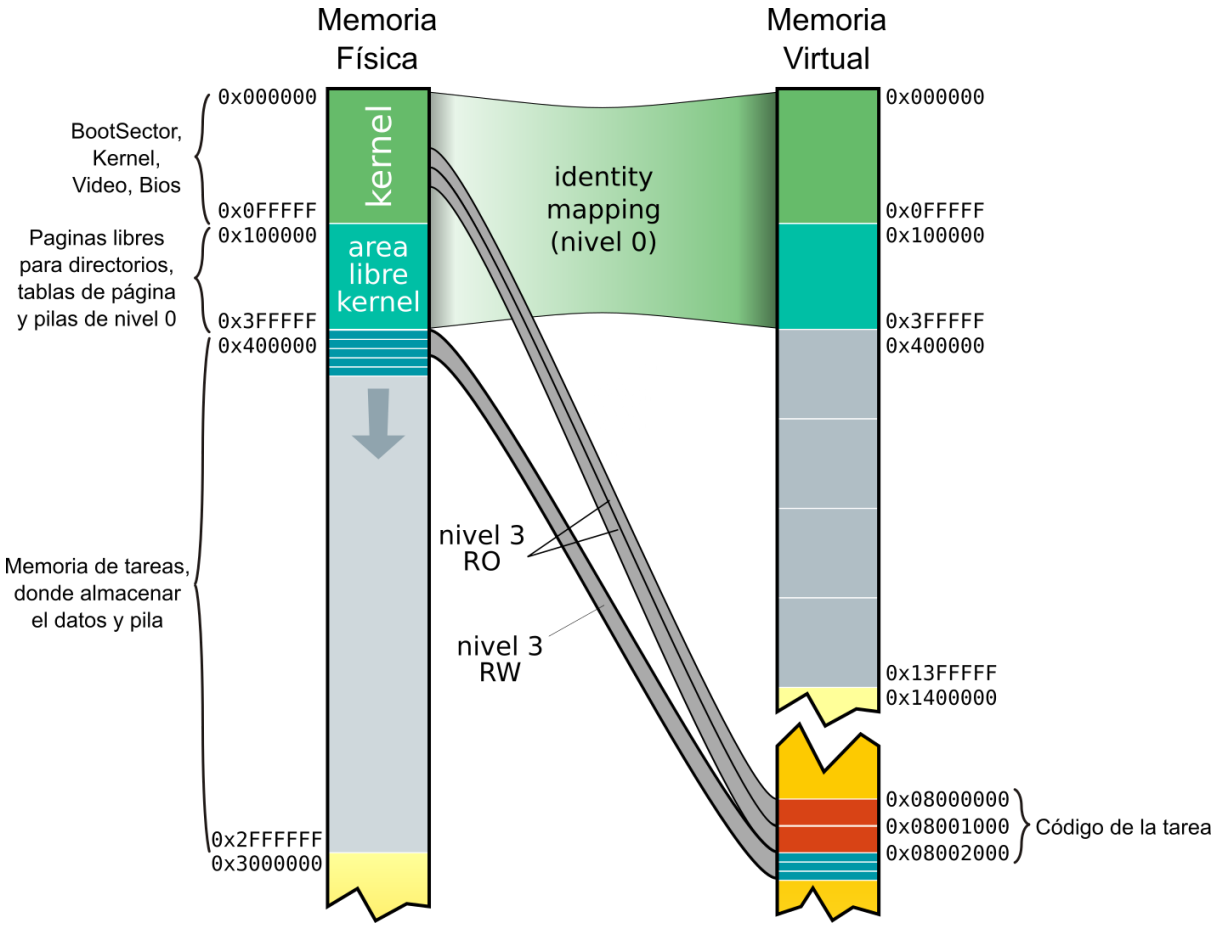
El taller de hoy: Tareas

Vamos a continuar trabajando con el kernel que estuvimos programando en los talleres anteriores. La idea es incorporar la posibilidad de ejecutar algunas tareas específicas. Para esto vamos a precisar:

- Definir las estructuras con las tareas que va a poder ejecutar
- Tener un scheduler que determine a qué tarea le toca ejecutase por un período de tiempo e ir intercambiarla por otra para compartir la CPU
- Iniciar el kernel con una *tarea inicial* y tener una *tarea idle* para cuando no haya tareas en ejecución

Recordamos el mapeo de memoria con el que venimos trabajando dado que las tareas que vamos a crear en este taller van a ser parte de esta organización de la memoria:





Archivos provistos

A continuación les pasamos la lista de archivos que forman parte del taller de hoy junto con su descripción:

- **`a20.asm`** - rutinas para habilitar y deshabilitar A20.
- **`Makefile`** - encargado de compilar y generar la imagen del floppy disk.
- **`types.h`** - declaraciones de tipos
- **`bochsrc, bochsdbg`** - configuración para inicializar Bochs.
- **`diskette.img`** - la imagen del floppy que contiene el boot-sector preparado para cargar el kernel.
- **`kernel.asm`** - esquema básico del código para el kernel.
- **`defines.h, colors.h`** - constantes y definiciones.
- **`gdt.h, gdt.c`** - definición de la tabla de descriptores globales.
- **`screen.h, screen.c`** - rutinas para pintar la pantalla.
- **`print.mac`** - macros útiles para imprimir por pantalla y transformar valores.
- **`idle.asm`** - código de la tarea Idle.
- **`syscall.h`** - interfaz a utilizar desde las tareas para los llamados al sistema.
- **`kassert.h`** - rutinas para garantizar invariantes en el kernel.
- **`i386.h`** - funciones auxiliares para utilizar assembly desde C.
- **`pic.c, pic.h`** - funciones pic_enable, pic_disable, pic_finish1 y pic_reset.
- **`isr.h, isr.asm`** - rutinas de atención de interrupciones
- **`idt.h, idt.c`** - definición de la tabla de interrupciones

- **`mmu.h, mmu.c`** - rutinas para mapeo de memoria
- **`tss.h, tss.c`** - definición de estructuras para el manejo de tareas
- **`sched.h, sched.c`** - scheduler de tareas del kernel
- **`task.h, task.c, taskA.c, taskB.c`** - código de las tareas

Ejercicios

1. Si queremos definir un sistema que use sólo dos tareas, ¿qué estructuras, cantidad de nuevas entradas por estructura y registros tenemos que configurar? Indicar el detalle de los bits de cada estructura, ¿qué formato tienen? y ¿dónde se encuentran almacenadas dichas estructuras?

2. ¿A qué llamamos cambio de contexto, cuándo se produce y qué efecto tiene sobre los registros del procesador? Expliquen en sus palabras que almacena el registro **TR** y cómo obtiene la información necesaria para ejecutar la tarea durante un cambio de contexto.

3. Al momento de realizar un cambio de contexto el procesador va almacenar el estado actual de acuerdo al selector almacenado en el registro **TR** y ha de restaurar aquel almacenado en la tss cuyo selector se asigna en el jmp far. ¿Qué consideraciones deberíamos tener para poder realizar el primer cambio de contexto? ¿Y cuáles cuando no tenemos tareas que ejecutar o se encuentran todas suspendidas?

4. ¿Qué hace el scheduler de un Sistema Operativo? ¿A qué nos referimos con que usa una política?

5. En un sistema de una única CPU, ¿cómo se hace para que lxs usuarixs vean todos los programas corriendo en simultáneo?

Primer Checkpoint

6. Observen en **tss.c** la definición de la Tarea Inicial

7. Ahora, vamos a agregar el *TSS Descriptor* correspondiente a la **tarea Idle** en la **GDT**.

a) Observen qué hace el método: **tss_gdt_entry_for_task**
b) Completen la entrada de la GDT correspondiente a la tarea Idle en el método **tss_init** de **tss.c**.

8. La primera tarea que va a ejecutar el procesador cuando arranque va a ser la **tarea Inicial**. Se encuentra definida en tss.c y que tiene todos sus campos en 0. Completen lo necesario en **kernel.asm** para que cargue como la tarea inicial. Recuerden que tenemos que cargar el registro **TR** (Task Register) con la instrucción **LTR**, la primera vez.

9. Una vez que el procesador comenzó su ejecución en la **tarea Inicial**, le vamos a pedir que salte a la **tarea Idle** con un **JMP**. Para eso, completar en **kernel.asm** el código necesario para saltar intercambiando **TSS**, entre la tarea inicial y la tarea Idle.

10. Agreguen dos breakpoints, uno antes del cambio de contexto entre la **tarea Inicial** y otro después de haber cargado la tarea Idle, en **idle.asm**, por ejemplo. Verifique los registros de propósito general

y el valor del **TR** en ambos para entender el cambio de contexto. También, verifiquen los valores de los registros **CR3** y registros de segmento **CS, DS**, **SS**. ¿Por qué hace falta tener definida la pila de nivel 0 en la tss?
Pd: Pueden usar el método **sreg** y **creg** para observar registros de segmento y de control. También es útil usar **info tss**

Segundo Checkpoint

11. Completar la función **tss_create_user_task** para que inicialice una TSS con los datos correspondientes a una tarea cualquiera. La función recibe por parámetro la dirección del código de una tarea y será utilizada más adelante para crear tareas. El código de las tareas se encuentra a partir de la dirección 0x00018000 ocupando dos páginas de 4 KB cada una según indica la figura 1. Para la dirección de la pila se debe reservar una página física sin usar, la misma crecerá desde la base de la pila. Tener en cuenta:

a) Para el mapa de memoria se debe construir uno nuevo utilizando la función **mmu_init_task_dir** implementada en el taller anterior**.** Este método va a crear su directorio de página y tablas de nivel 3. Dejando en la dirección virtual **TASK_CODE_VIRTUAL** el código y en **TASK_STACK_BASE**, la pila.
b) Además, tener en cuenta que cada tarea utilizará una pila distinta de nivel 0, para esto se debe pedir una nueva página del área libre de kernel a tal fin. Pueden usar el método: **mmu_next_free_kernel_page**.

12. Observen con **info gdt** las entradas correspondientes a los descriptores de las tss que cargaron tanto como para tareas de usuario, como las tareas idle e inicial.

Tercer Checkpoint

13. Estando definidas **sched_task_offset** y **sched_task_selector**:

```
sched_task_offset:      dd 0xFFFFFFFF
sched_task_selector:    dw 0xFFFF
```

Y siendo la siguiente es una implementación de una interrupción del reloj:

```
global _isr32

_isr32:
      pushad
      call pic_finish1
```

```
call sched_next_task

str cx
cmp ax, cx
je .fin

mov word [sched_task_selector], ax
jmp far [sched_task_offset]

.fin:
popad
iret
```

a) Expliquen con sus palabras que se estaría ejecutando en cada tic del reloj línea por línea

14. En el archivo **isr.asm**, se encuentra la interrupción de reloj.

a) Comenten la línea necesaria para llamar al scheduler obteniendo la próxima tarea a ejecutar.

b) En:

```
jmp far [sched_task_offset]
```

¿De que tamaño es el dato que estaría leyendo desde la memoria? ¿Qué indica cada uno de estos valores? ¿Tiene algún efecto el offset elegido?

c) ¿A dónde regresa la ejecución (eip) de una tarea cuando vuelve a ser puesta en ejecución?

15. Compilen y observen la ejecución de las tareas.

16. En los archivos **sched.c** y **sched.h** se encuentran definidos los métodos necesarios para el Scheduler. Expliquen cómo funciona el scheduler, es decir, cómo decide cuál es la próxima tarea a ejecutar. Pueden encontrarlo en la función **sched_next_task**.

Cuarto Checkpoint

17. Revisen los archivos **idt.c, idt.h, isr.asm, syscall.h, mmu.h, mmu.c** y expliquen qué función cumplen los sys calls **syscall_getid, syscall_print, syscall_print_dec** y **syscall_print_hex**.

18. ¿Por qué implementamos la sys call **syscall_print** en lugar de llamar directamente a la función **print** de **screen.c**?

19. Proveemos la implementación de las sys calls **vaddr_t syscall_getshm(uint8_t mem_id)** y **void syscall_freeshm(uint8_t mem_id)** que le permiten (**syscall_getshm**) a una tarea pedir acceso a una página de memoria compartida a partir de su identificador **mem_id**, la sys call devolverá la dirección

virtual a partir de la cual la tarea puede acceder a esa página, **syscall_freeshm** indica que la tarea ya no accederá a la página de memoria compartida. En **mmu.c** se encuentran las implementaciones de kernel de estas sys call (**mmu_getshm** y **mmu_freeshm**). ¿Cómo decide el kernel a qué dirección virtual mapear la página compartida? ¿Cómo decide a qué página física referir y cuándo lo hace?

20. Como parte de la inicialización del kernel, en kernel.asm se llama a la función task_init de task.c que a su vez llama a create_task. Observe las siguientes líneas:

```
size_t gdt_id = (tipo == TASK_A? GDT_IDX_TASK_A_START :
          GDT_IDX_TASK_B_START) + indice;
gdt[gdt_id] = tss_gdt_entry_for_task(&tss_tasks[task_id]);
sched_add_task(gdt_id << 3, task_id);
```

¿Qué está haciendo la función **tss_gdt_entry_for_task**? ¿Por qué motivo se realiza el desplazamiento a izquierda de **gdt_id** al pasarlo como parámetro de **sched_add_task**?

21. Explique en sus palabras qué está haciendo el código de **taskA.c**.

22. Explique en sus palabras qué está haciendo el código de **taskB.c**.

23. ¿Por qué debemos dejar a las tareas ejecutando la instrucción **nop** indefinidamente luego de terminar de procesar?

Quinto Checkpoint

24. [Opcional] Resuman con su equipo todas las estructuras vistas desde el Taller 1 al Taller 4. Escriban el funcionamiento general de segmentos, interrupciones, paginación y tareas en los procesadores Intel. ¿Cómo interactúan las estructuras? ¿Qué configuraciones son fundamentales a realizar? ¿Cómo son los niveles de privilegio y acceso a las estructuras?