

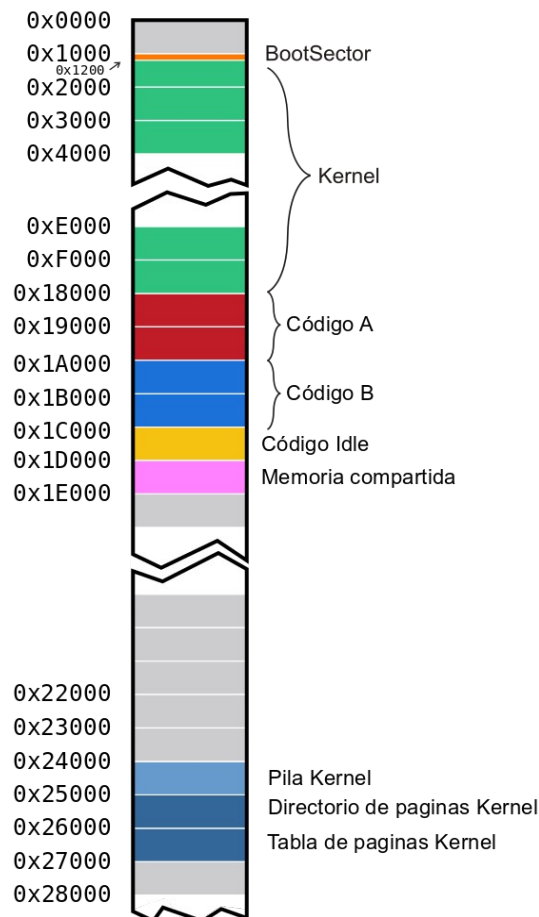
Organización del Computador II
System Programming
Taller 4: Tareas

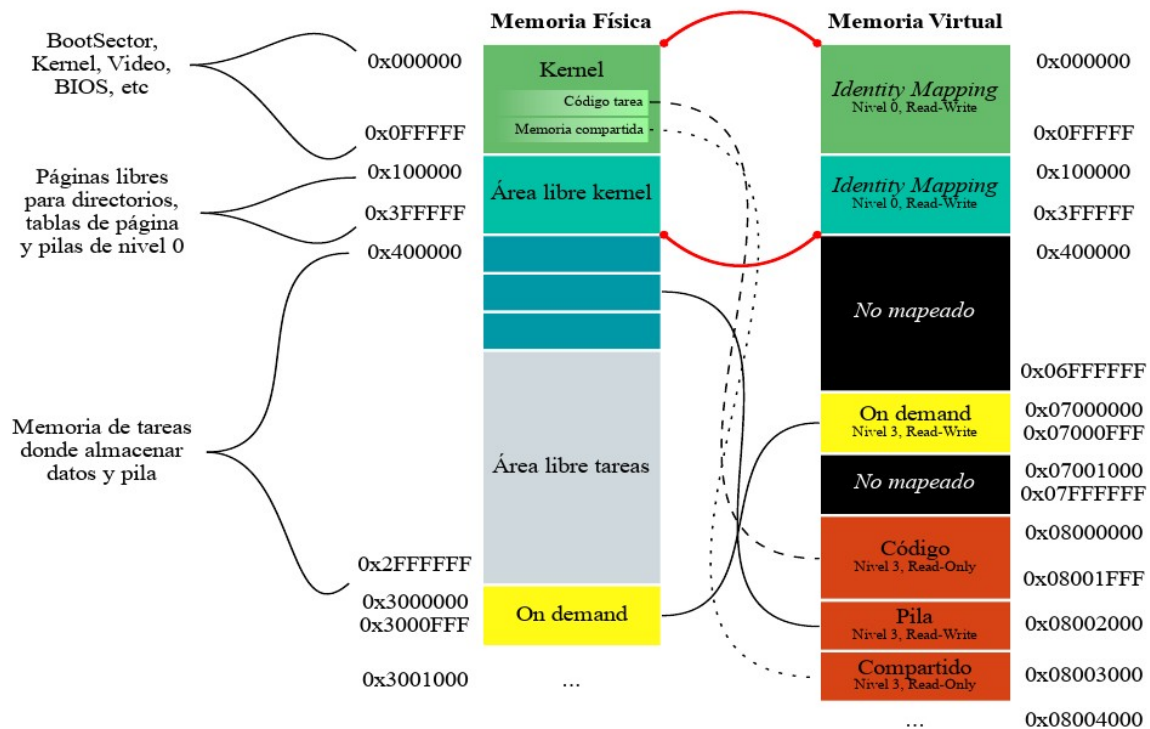
El taller de hoy: Tareas

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 de las tareas disponibles para ser ejecutadas
- Tener un scheduler que determine la tarea a la que le toca ejecutarse en un período de tiempo, y el mecanismo para el intercambio de tareas de 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. 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:

- **Makefile** - encargado de compilar y generar la imagen del floppy disk.
- **idle.asm** - código de la tarea Idle.
- **shared.h** - estructura de la página de memoria compartida
- **tareas/syscall.h** - interfaz para realizar llamadas al sistema desde las tareas
- **tareas/task_lib.h** - Biblioteca con funciones útiles para las tareas
- **tareas/task_prelude.asm** - Código de inicialización para las tareas
- **tareas/taskPong.c** - código de la tarea que usaremos
(**tareas/taskGameOfLife.c**, **tareas/taskSnake.c**, **tareas/taskTipear.c** - código de otras tareas de ejemplo)
- **tareas/taskPongScoreboard.c** - código de la tarea que deberán completar
- **tss.h**, **tss.c** - definición de estructuras y funciones para el manejo de las TSSs
- **sched.h**, **sched.c** - scheduler del kernel
- **tasks.h**, **tasks.c** - Definición de estructuras y funciones para la administración de tareas
- **isr.asm** - Handlers de excepciones y interrupciones (en este caso se proveen las rutinas de atención de interrupciones)
- **task_defines.h** - Definiciones generales referente a tareas

Ejercicios

1. Si queremos definir un sistema que utilice sólo dos tareas, ¿Qué nuevas estructuras, cantidad de nuevas entradas en las estructuras ya definidas, y registros tenemos que configurar? ¿Qué formato tienen? ¿Dónde se encuentran almacenadas?
2. ¿A qué llamamos cambio de contexto? ¿Cuándo se produce? ¿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 una tarea después de un cambio de contexto.
3. Al momento de realizar un cambio de contexto el procesador va almacenar el estado actual de acuerdo al selector indicado 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 los programas parezcan ejecutarse en simultáneo?
6. En **tss.c** se encuentran definidas las TSSs de la Tarea **Inicial** e **Idle**. Ahora, vamos a agregar el TSS *Descriptor* correspondiente a estas tareas en la **GDT**.
 - a) Observen qué hace el método: **tss_gdt_entry_for_task**
 - b) Escriban el código del método **tss_init** de **tss.c** que agrega dos nuevas entradas a la **GDT** correspondientes al descriptor de TSS de la tarea Inicial e Idle.
 - c) En **kernel.asm**, luego de habilitar paginación, agreguen una llamada a **tss_init** para que efectivamente estas entradas se agreguen a la **GDT**.
 - d) Correr el *qemu* y usar **info gdt** para verificar que los descriptores de tss de la tarea Inicial e Idle estén efectivamente cargadas en la GDT
7. Como vimos, la primer tarea que va a ejecutar el procesador cuando arranque va a ser la **tarea Inicial**. Se encuentra definida en **tss.c** y tiene todos sus campos en 0. Antes de que comience a ciclar infinitamente, 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. Previamente llamar a la función `tasks_screen_draw` provista para preparar la pantalla para nuestras tareas.

Si obtienen un error, asegurense de haber proporcionado un selector de segmento para la tarea inicial. Un selector de segmento no es sólo el índice en la GDT sino que tiene algunos bits con privilegios y el *table indicator*.
8. 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.
9. Utilizando **info tss**, verifiquen el valor del **TR**. También, verifiquen los valores de los registros **CR3**

con **creg** y registros de segmento **CS**, **DS**, **SS** con **sreg**. ¿Por qué hace falta tener definida la pila de nivel 0 en la tss?

10. En **tss.c**, 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.

Las direcciones físicas del código de las tareas se encuentran en **defines.h** bajo los nombres **TASK_A_CODE_START** y **TASK_B_CODE_START**.

El esquema de paginación a utilizar es el que hicimos durante el taller anterior. Tener en cuenta que cada tarea utilizará una pila distinta de nivel 0.

Primer Checkpoint

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

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

Y siendo la siguiente 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
- b) En la línea que dice **jmp far [sched_task_offset]** ¿De qué 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?

12. Para este Taller la cátedra ha creado un scheduler que devuelve la próxima tarea a ejecutar.

- a) En los archivos **sched.c** y **sched.h** se encuentran definidos los métodos necesarios para el Scheduler. Expliquen cómo funciona el mismo, es decir, cómo decide cuál es la próxima tarea a ejecutar. Pueden encontrarlo en la función **sched_next_task**.
- b) Modifiquen **kernel.asm** para llamar a la función **sched_init** luego de iniciar la TSS
- c) Compilen, ejecuten *qemu* y vean que todo sigue funcionando correctamente.

Segundo Checkpoint

14. Como parte de la inicialización del kernel, en `kernel.asm` se pide agregar una llamada a la función `tasks_init` de `task.c` que a su vez llama a `create_task`. Observe las siguientes líneas:

```
int8_t task_id = sched_add_task(gdt_id << 3);
tss_tasks[task_id] = tss_create_user_task(task_code_start[tipo]);
gdt[gdt_id] = tss_gdt_entry_for_task(&tss_tasks[task_id]);
```

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

15. Ejecuten las tareas en `qemu` y observen el código de estas superficialmente.

- (a) ¿Qué mecanismos usan para comunicarse con el kernel?
- (b) ¿Por qué creen que no hay uso de variables globales? ¿Qué pasaría si una tarea intentase escribir en su ``.data`` con nuestro sistema?
- (c) Cambien el divisor del PIT para "acelerar" la ejecución de las tareas:

```
; El PIT (Programmable Interrupt Timer) corre a 1193182Hz.
; Cada iteracion del clock decrementa un contador interno, cuando éste llega
; a cero se emite la interrupción. El valor inicial es 0x0 que indica 65536,
; es decir 18.206 Hz
mov ax, DIVISOR
out 0x40, al
rol ax, 8
out 0x40, al
```

16. Observen `tareas/task_prelude.asm`. El código de este archivo se ubica al principio de las tareas.

- a. ¿Por qué la tarea termina en un loop infinito?
- b. [Opcional] ¿Qué podríamos hacer para que esto no sea necesario?

Tercer Checkpoint

Ahora programaremos nuestra tarea. La idea es disponer de una tarea que imprima el `score` (puntaje) de todos los Pongs que se están ejecutando. Para ello utilizaremos la memoria mapeada *on demand* del taller anterior.

Análisis:

18. Analicen el `Makefile` provisto. ¿Por qué se definen 2 “tipos” de tareas? ¿Como harían para ejecutar una tarea distinta? Cambien la tarea `Snake` por una tarea `PongScoreboard`.

19. Mirando la tarea `Pong`, ¿En que posición de memoria escribe esta tarea el puntaje que queremos imprimir? ¿Cómo funciona el mecanismo propuesto para compartir datos entre tareas?

Programando:

20. Completen el código de la tarea `PongScoreboard` para que imprima en la pantalla el puntaje de todas las instancias de `Pong` usando los datos que nos dejan en la página compartida.

Cuarto Checkpoint

- 21.** [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?
- 22.** [Opcional] ¿Qué pasa cuando una tarea dispara una excepción? ¿Cómo podría mejorarse la respuesta del sistema ante estos eventos?