Práctica de paginación

Organización del Computador II

En este taller vamos a inicializar y a habilitar los mecanismos de manejo de memoria de nuestro kernel. En particular, activaremos el módulo de paginación del microprocesador

1. Organización de la memoria

Primero vamos a explicar cómo se encuentra el mapa de memoria física para comprender de qué modo inicializar las tablas de memoria. El primer MB de memoria física será organizado según indica la figura 1. En la misma se observa que a partir de la dirección `0x1200` se encuentra ubicado el *kernel*; inmediatamente después se ubica el código de las tareas A y B. A continuación se encuentra el código de la tarea Idle. El resto del mapa muestra el rango para la pila del kernel, desde `0x24000` y a continuación la tabla y directorio de páginas donde inicializar paginación para el kernel. La parte derecha de la figura muestra la memoria a partir de la dirección `0xA0000`, donde se encuentra mapeada la memoria de vídeo y el código del `BIOS`.

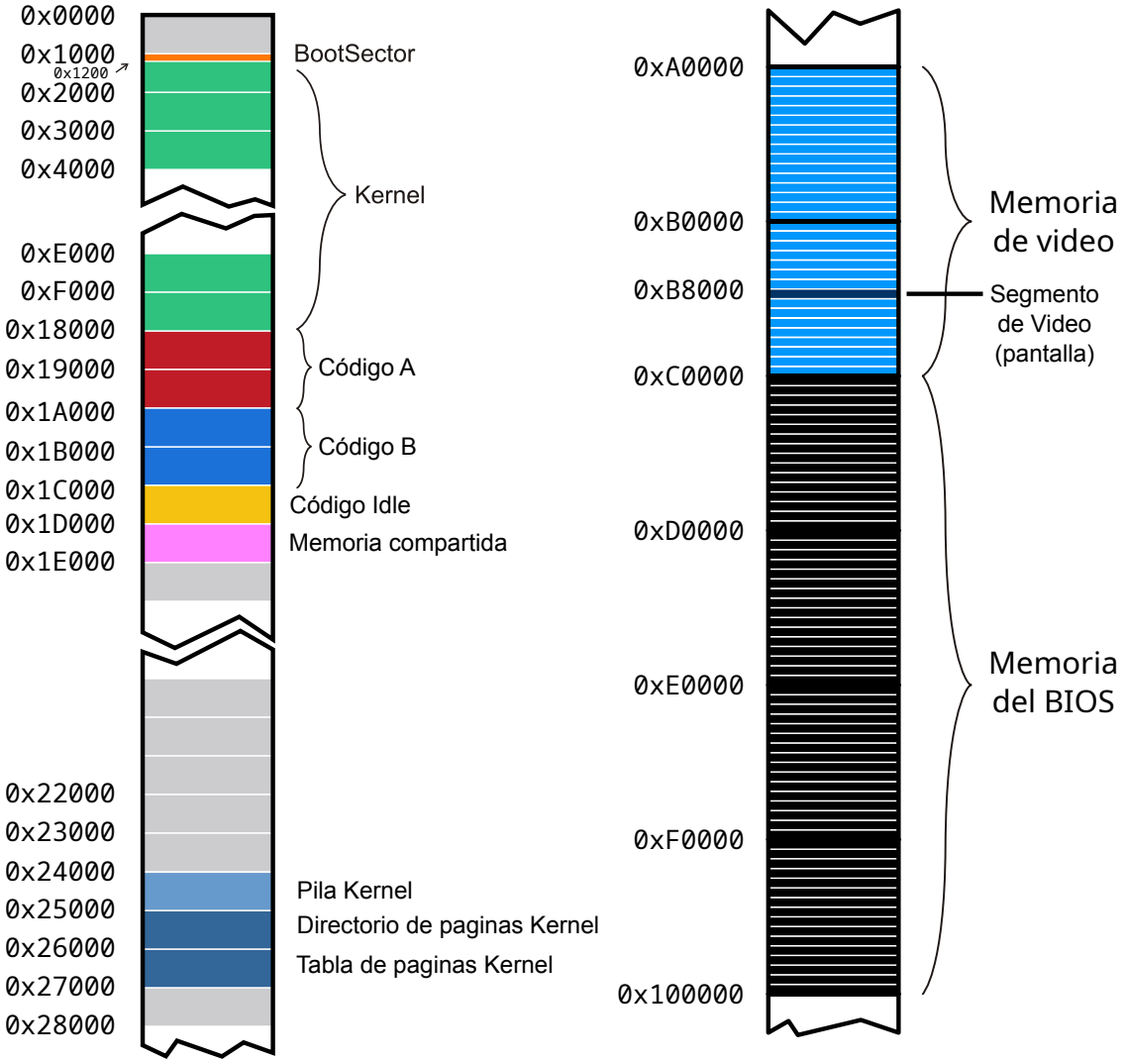

Figura 1: Mapa de la organización de la memoria física del *kernel*

2. Unidad de Manejo de Memoria

Ya viendo la memoria física en forma más general, vemos que se divide en: *kernel*, *área libre kernel* y *área libre tareas*.

El área asociada al *kernel* corresponde al primer MB de memoria, el *área libre kernel* a los siguientes 3 MB, y el *área libre tareas* comienza en el cuarto MB de memoria.

La administración de las áreas libres de memoria (área libre de kernel y área libre de tareas) se realizará a partir de una región de memoria específica para cada una. Podemos comprenderlas como un arreglo predefinido de páginas y dos contadores de páginas, uno para kernel y otro para usuarix, que indican cuál será la próxima página a emplear de cada región. Para páginas de kernel el arreglo va de `0x100000` a `0x3FFFFF` y para páginas de usuarix de `0x400000` a `0x2FFFFFF`. Luego de cada pedido incrementamos el contador correspondiente. Para el contexto de la materia no implementamos un mecanismo que permita liberar las páginas pedidas. Vamos a referirnos al módulo que implementa este mecanismo como la **unidad de manejo de memoria**, o en inglés, *memory management unit*, MMU.

Las páginas del *área libre kernel* serán utilizadas para datos del kernel: directorios de páginas, tablas de páginas y pilas de nivel cero. Las páginas del *área libre tareas* serán utilizadas para datos de las tareas, stack de las mismas y memoria compartida bajo demanda.

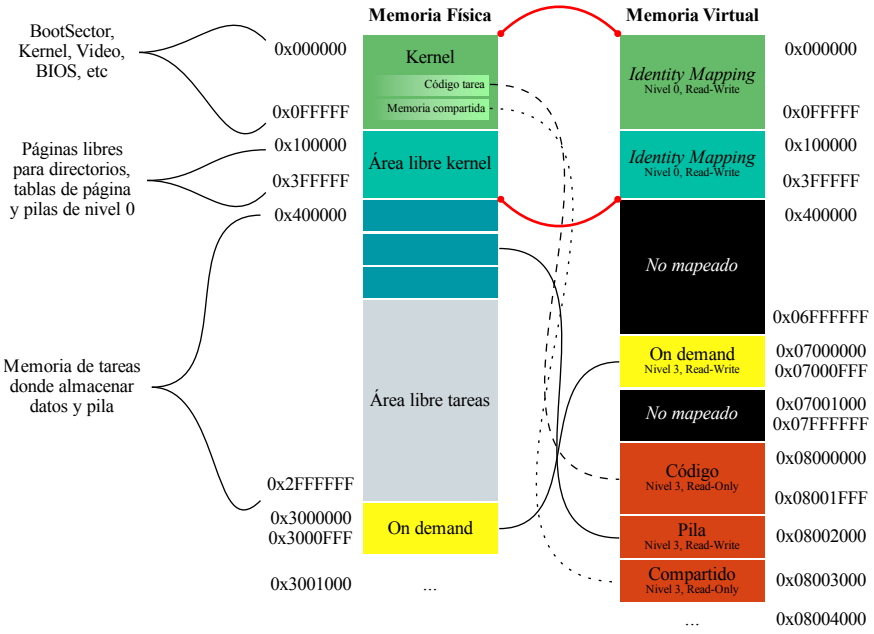

Figura 2: Mapa de memoria de la tarea

La memoria virtual de cada una de las tareas tiene mapeado inicialmente el *kernel* y el *área libre kernel* con *identity mapping* en nivel 0. Además se mapearán las páginas de código correspondientes, la página de memoria compartida y una página para la pila obtenida desde el *área libre de tareas*.

A partir de la dirección virtual *0x07000000* se encontrará una página de memoria compartida de lectura y escritura nivel 3. Esta página será mappeada on-demand cuando se intente hacer una lectura o escritura en este espacio. La página física debe encontrarse en la dirección *0x03000000*.

El código de las tareas se encontrará a partir de la dirección virtual *0x08000000* y será mapeado como sólo lectura de nivel 3 a la dirección física del código correspondiente. Cada tarea puede utilizar hasta 8Kb de código. El stack será mapeado en la página siguiente, con permisos de lectura y escritura. La página física debe obtenerse del *área libre de tareas*. Finalmente, luego de la pila se mapeará la página de memoria compartida como sólo lectura de nivel 3. Esta página la usaremos más adelante para que las tareas puedan acceder fácilmente a información relevante del contexto de ejecución (teclas apretadas, tiempo desde el arranque del sistema, etc).

Notas

- Por construcción del *kernel*, las direcciones de los mapas de memoria (`page directory` y `page table`) están mapeadas con *identity mapping*.
- En las funciones en donde se modifica el directorio o tabla de páginas, se debe llamar a la función `tlbflush` para que se invalide la *cache* de traducción de direcciones.

Uso de qemu+gdb

A continuación uns descripción de los comandos que pueden utilizar en gdb para acceder a la información sobre el mapeo de páginas:

- `info page`
  muestra información general sobre el mapeo

- `info page` [vaddr]
  muestra la traducción de *vaddr* dando el detalle de las entradas PD y PT correspondientes

- `info page directory`
  lista las entradas presentes del PD actual con sus atributos

- `info table [idx]`
  lista las entradas presentes de la i-ésima page table

- `x /nuf [addr]`
  Muestra el contenido de la dirección `[addr]`

- `xp /nuf [addr]`
  Muestra el contenido de la dirección física `[addr]`
  `n` es el número que indica cuantos valores se mostrarán (default 1)
  `u` es el tamaño de la unidad, puede ser[1]:

```
b : byte                h : word (half-word)
w : doubleword(word)    g : quadword(giant word)
```

  `f` es el formato del número, puede ser:

```
x : hex      d : decimal       u : sin signo
o : octal    t : binario       c : char
s : ascii    i : instrucción
```

3. Ejercicios

a) ¿Cuántos niveles de privilegio podemos definir en las estructuras de paginación?

b) ¿Cómo se traduce una dirección lógica en una dirección física? ¿Cómo participan la dirección lógica, el registro de control `CR3`, el directorio y la tabla de páginas? Recomendación: describan el proceso en pseudocódigo

c) ¿Cuál es el efecto de los siguientes atributos en las entradas de la tabla de página?

- `D`
- `A`
- `PCD`
- `PWT`
- `U/S`
- `R/W`
- `P`

d) ¿Qué sucede si los atributos U/S y R/W del directorio y de la tabla de páginas difieren? ¿Cuáles terminan siendo los atributos de una página determinada en ese caso? Hint: buscar la tabla *Combined Page-Directory and Page-Table Protection* del manual 3 de Intel

e) Suponiendo que el código de la tarea ocupa dos páginas y utilizaremos una página para la pila de la tarea. ¿Cuántas páginas hace falta pedir a la unidad de manejo de memoria para el directorio, tablas de páginas y la memoria de una tarea?

f) Completen las entradas referentes a MMU de `defines.h` y comprendan la función y motivación de dichos defines:

- `VIRT_PAGE_OFFSET(X)` devuelve el offset asociado a una dirección virtual `X`
- `VIRT_PAGE_TABLE(X)` devuelve la tabla de páginas asociada a una dirección virtual `X`
- `VIRT_PAGE_DIR(X)` devuelve el directorio asociado a una dirección virtual `X`
- `CR3_TO_PAGE_DIR(X)` obtiene la dirección física del directorio donde `X` es el contenido del registro `CR3`.

[1] Entre paréntesis el nombre consistente con gdb

- `MMU_ENTRY_PADDR(X)` obtiene la dirección física correspondiente, donde `X` es el campo address de 20 bits en una entrada de la tabla de páginas o del page directory

g) ¿Qué es el buffer auxiliar de traducción (*translation lookaside buffer* o **TLB**) y por qué es necesario purgarlo (`tlbflush`) al introducir modificaciones a nuestras estructuras de paginación (directorio, tabla de páginas)? ¿Qué atributos posee cada traducción en la TLB? Al desalojar una entrada determinada de la TLB ¿Se ve afectada la homóloga en la tabla original para algún caso?

Checkpoint 1

a) Escriban el código de las funciones `mmu_next_free_kernel_page`, `mmu_next_free_user_page` y de `mmu_init_kernel_dir` de `mmu.c` para completar la inicialización del directorio y tablas de páginas para el *kernel*.

Recuerden que las entradas del directorio y la tabla deben realizar un mapeo por identidad (las direcciones lineales son iguales a las direcciones físicas) para el rango reservado para el kernel, de `0x00000000` a `0x003FFFFF`, como ilustra la figura 2. Esta función debe inicializar también el directorio de páginas en la dirección `0x25000` y las tablas de páginas según muestra la figura 1. ¿Cuántas entradas del directorio de página hacen falta?

b) Completar el código para activar paginación en `kernel.asm`. Recuerden que es necesario inicializar el registro `CR3` y activar el bit correspondiente de CR0. Esta inicialización debe realizarse antes de activar las interrupciones del procesador.

c) Introduzcan un breakpoint luego de activar paginación y prueben hacer **info page** para comprobar que el mapeo identidad se realizó correctamente.

Checkpoint 2

a) Completen el código de la función `mmu_map_page`, `mmu_unmap_page`

b) Completen el código de `copy_page`, ¿por qué es necesario mapear y desmapear las páginas de destino y fuente? ¿Qué función cumplen `SRC_VIRT_PAGE` y `DST_VIRT_PAGE`? ¿Por qué es necesario obtener el CR3 con rcr3()?

c) Realicen una prueba donde se compruebe el funcionamiento de `copy_page`. Pueden usar gdb con el comando `x` para inspeccionar el contenido de direcciones virtuales y `xp` para inspeccionar el contenido de direcciones físicas.

d) Completen la rutina (`mmu_init_task_dir`) encargada de inicializar un directorio de páginas y tablas de páginas para una tarea, respetando la figura 2. La rutina debe mapear las páginas de código como solo lectura, a partir de la dirección virtual `0x08000000`, el stack como lectura-escritura con base en `0x08003000` y la página de memoria compartida luego del stack. Recuerden que la memoria para la pila de la tarea debe obtenerse del area libre de tareas.

e) Completen la rutina de atención de interrupción del Page Fault para que, si se intenta acceder al rango de memoria compartido on demand cuando este no está mapeado, se mapee. Respeten la figura 2. Se debe mapear como de lectura-escritura a nivel usuario.

f) A modo de prueba, en `kernel.asm` vamos a construir un mapa de memoria para una tarea ficticia (es decir, cargar el `CR3` de una tarea) e intercambiarlo con el del *kernel*. Para esto tendrán que usar la función antes construida, `mmu_init_task_dir`. Supongan que la tarea se encuentra ubicada en la dirección física 0x18000.

- Una vez hecho el cambio de cr3, hagan dos escrituras en alguna parte de la zona de memoria compartida on-demand y luego vuelvan a la normalidad. Deberían ver el mensaje `''Atendiendo Page Fault...''` luego de la primer escritura y ningún mensaje luego de la segunda.
- Inspeccionen el mapa de memoria con el comando `info page` con breakpoints una vez que se asigna el `CR3` de la tarea y cuando se restituye el `CR3` del kernel.

Checkpoint 3