DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Batalla Bytal

Organización del Computador II

Segundo Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Travi, Fermín	234/13	fermintravi@gmail.com
Benzo, Mariano	198/14	marianobenzo@gmail.com
Martinez Quispe, Franco	025/14	francogm01@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

1. Detalles de la implementación

Para la implementación del kernel, se hizo uso de varios archivos auxiliares que se encargan de una porción en particular de la implementación. Ya que cada archivo representa una clase distinta, los atributos propios de cada archivo se mantuvieron privados dentro de la clase; es decir, la única manera de modificar los atributos es a través de métodos de la clase.

La única variable global que es accedida por todas las clases es la variable *corriendoBandera*, la cual indica si se encuentra la función de alguna bandera en ejecución. Se implementó de esta manera para permitir escribir directamente sobre ella al momento de llamar a Int 0x66.
Por lo demás, se mantuvo una concordancia con respecto al nombre de la clase y los métodos que implementan. Por ejemplo, *screen.c* contiene todos los métodos para imprimir en pantalla y ninguna otra clase posee métodos que impriman en pantalla.

1.1. kernel.asm

El archivo *kernel.asm* sigue los comentarios escritos por la cátedra sobre los pasos a seguir:

```
; pintar pantalla, todos los colores, que bonito!
call pintar_scheduler
call pintar_buffer_estado
call pintar_buffer_mapa
```

En este caso, se pinta el modelo de pantalla según las imágenes del enunciado en el *buffer* de memoria asignado para el modo estado y modo mapa, cuyas direcciones de memoria están especificadas en el enunciado. Dichos métodos se encuentran en *screen.c*.

```
; inicializar el manejador de memoria
call mmu_inicializar_dir_kernel
```

En este caso, el método *mmu_inicializar_dir_kernel()* se encarga de crear el directorio de páginas para el kernel, donde se realiza *identity mapping* sobre los primeros 7.5 MB de memoria. Luego, se realiza un mapeo de la dirección 0x40000000 a la dirección física donde se encuentra el código de la tarea *idle*. Ya que *mmu.c* se encarga de construir los espacios de memoria, los métodos se encuentran allí.

El resto de los comentarios no poseen ninguna modificación especial, excepto en este caso:

```
; inicializar memoria de tareas
; inicializar tarea idle
; inicializar todas las tsss
; inicializar entradas de la gdt de las tsss
call tss_inicializar
```

El método *tss_inicializar()* se encarga de realizar todos los comentarios descriptos anteriormente. Es decir, de inicializar la memoria de tareas, incluyendo la tarea idle, sus respectivas estructuras de las tss y las entradas en la GDT con el puntero a dichas estructuras. En este caso, se rompe el invariante descrito al principio, ya que un método de *tss.c* modifica directamente una estructura del archivo *gdt.c*. Sin embargo, se hizo de esta manera porque era necesario primero crear la estructura para luego generar su puntero y escribirlo en la GDT, lo cual no podía realizarse directamente en *gdt.c*.

1.2. gdt.c

Este archivo no posee ninguna particularidad, ya que simplemente se encarga de escribir sobre la estructura gdt_descriptor provista por la cátedra. Las entradas se completan según lo visto en clase, a

diferencia de las entradas correspondientes a las tsss, cuyos campos *base* se fijaron en 0x00 para luego ser completados en el método *tss_inicializar()*, como se mencionó anteriormente.

Para facilitar el acceso a los índices, en el archivo *defines.h* se definieron constantes con los índices correspondientes a cada una de las entradas de la tabla.

1.3. idt.c

idt.c se encarga de manejar cualquier cosa relacionado con interrupciones (junto con el archivo *isr.asm*). El método *idt_inicializar()* completa la tabla IDT según lo visto en clase y utilizando la estructura provista por la cátedra.

El método *atender_int(interrupcion, registros)* se encarga de escribir en pantalla, mediante métodos de *screen.c*, la excepción ocurrida (junto con la información pertinente) y de invocar a métodos de *sched.c* para desalojar la tarea que haya provocado la excepción.

El método *int_teclado(makeCode)* se encarga de manejar las interrupciones de teclado.

Luego, los métodos *anclar* y *navegar* se encargan de realizar lo pedido según los servicios del sistema que llaman las tareas.

1.4. isr.asm

Dado que es más simple manejar las interrupciones desde C, este archivo se encarga de llamar a los métodos correspondientes (junto con sus parámetros) en *idt.c*. La única excepción ocurre con la interrupción del reloj, la cual llama a métodos del *scheduler*, ubicados en *sched.c*. Para facilitar su manejo, el método invocado retorna el descriptor de segmento al cual hay que utilizar en el jump far:

```
_isr32:
    pushad

    call screen_proximo_reloj
    call proximo_indice
    cmp ax, 0
    je .noJump
     mov [selector], ax
     call fin_intr_pic1
     jmp far [offset]
     jmp .fin
    .noJump:
     call fin_intr_pic1
    .fin:
     popad
    iret
```

El resto de las interrupciones simplemente invocan a los métodos en *idt.c*, según corresponda. La diferencia ocurre con la interrupción 0x66, donde el mismo archivo escribe sobre la variable global *corriendoBandera*.

```
_isrx66:
    pushad
    call fin_intr_pic1
    mov byte [corriendoBandera], 0x00
    popad
    jmp 0xB8:0x00       ; tarea_idle
    iret
```

1.5. mmu.c

Este archivo únicamente se encarga de generar los mapas de memoria, conteniendo métodos para generar el mapa de memoria del kernel (*mmu_inicializar_dir_kernel()* y para generar el mapa de memoria de una tarea determinada (*mmu_inicializar_dir_tarea(tarea, dir)*). Como la pantalla en modo *mapa* muestra dónde están mapeadas las páginas de las tareas, dentro de dicho método se invoca a la función *asignar_dir(tarea, dir, pagina)* de la clase *screen.c* para dibujarlas en el *buffer* del mapa.

Por otro lado, contiene una variable propia de la clase (*int* p) que contiene la dirección de memoria física donde se ubicará la próxima página. Luego, mediante el método *prox_pagina()* cualquier clase puede pedir un puntero a una página de memoria lista para ser escrita.

1.6. screen.c

Acá se encuentran todos los métodos pertinentes para imprimir en pantalla. Los métodos *pintar_buffer_mapa()*, *pintar_buffer_estado()* y *pintar_scheduler()* se encargan de generar una base sobre la cual se imprimirán datos a medida que se ejecutan las tareas.

En particular, *coordenadas(dir)* se encarga de convertir una dirección de memoria en una posición de la pantalla. Luego, es utilizado por la función *asignar_dir(tarea, dir, pagina)*, la cual se encarga de actualizar los datos de las tareas tanto en el *buffer* de estado como en el *buffer* de mapa.

Por otro lado, el método *redigir_misil(dir)* se encarga de dibujar a dónde fue dirigido el último misil y borra del mapa al anterior.

Ya que es necesario saber dónde se ubicaban en la arena las tareas y misiles previamente a ser modificados, la clase contiene una matriz que almacena las direcciones físicas donde están mapeadas las páginas de las tareas y se posee otra variable que indica la coordenada del último misil lanzado. De esta manera, se pueden borrar del mapa una vez que sus valores son modificados.

Se modificaron los métodos provistos por la cátedra para permitir indicarles sobre qué posición de memoria escribir. Además, al método *print_hex* se le agregó un parámetro para indicar si hay que imprimir el número con el prefijo de '0x' o no.

1.7. tss.c

Si bien este archivo simplemente se encarga de crear y completar las estructuras *tss* de las tareas, contiene algunas particularidades.

Por empezar, es aquí donde se asignan en la tabla GDT las direcciones *base* de las entradas de las *tss*, mediante el puntero a la estructura. Para realizar esto, se utilizan macros muy similares a los provistos por la cátedra para completar la tabla IDT.

Por otro lado, el método *tss_inicializar()* a la vez que completa los datos sobre las *tss* de las tareas, invoca al método descripto anteriormente *mmu_inicializar_dir_tarea(tarea, dir)*, el cual genera el mapa de memoria de la tarea especificada. De esta manera, es más fácil asignarle el puntero a donde se ubica el directorio de páginas de cada tarea (campo *cr3* de la estructura):

```
tss_navios[i].cr3    = mmu_inicializar_dir_tarea(i, mar);
```

1.8. sched.c

La implementación del *scheduler* contiene variables propias de la clase que indican qué tarea se está ejecutando (*int* currTask), qué bandera se está ejecutando (*int* currFlag), si corresponde ejecutar las banderas (*char* modoBandera) y cuánto falta para ejecutar las banderas (*char* cicloBandera). A su vez, posee dos arreglos (*tasks* y *flags*) de tamaño equivalente a la cantidad de tareas donde se almacenan los selectores de segmento para cada tarea o bandera, según corresponda.

El método más importante de la clase es *proximo_indice()*, que se encarga de devolver el selector de segmento de la tarea (o bandera) a la cual corresponda saltar:

```
unsigned short proximo_indice(){
    if (corriendoBandera){
        tasks[currFlag] = 0x00;
        flags[currFlag] = 0x00;
    }
    if (modoBandera){
        corriendoBandera = TRUE;
        return sched_proxima_bandera();
    }else{
        cicloBandera++;
        currFlag = -1;
        if (cicloBandera == 3)
            modoBandera = TRUE; // En el prox tick se ejecuta sched_proxima_bandera()
        return sched_proximo_indice();
    }
}
```

Al principio, se verifica si se estaba ejecutando una bandera. En caso de hacerlo, corresponde que no se vuelva a ejecutar dicha tarea, por lo cual se fija en 0 sus respectivos valores en los arreglos. Por invariante, si un selector de segmento dentro del arreglo es 0, se interpreta que la tarea no puede ser ejecutada y se la saltea.

Luego, se fija si corresponde ejecutar las banderas, por lo cual fija en *true* la variable global *corriendoBandera* (que indica si se está ejecutando una bandera) y llama al método *sched_proxima_bandera()* que retorna el selector de segmento de la próxima bandera a ejecutar.

En caso de que no haya que correr las banderas, se aumenta el contador *cicloBandera* y se setea en el valor inicial a *currFlag*. Si ya se ejecutaron tres tareas seguidas, se activa el *modoBandera* para ejecutar las banderas en la próxima interrupción de reloj. Una vez hecho eso, se invoca al método *sched_proximo_indice()* que retorna el selector de segmento de la próxima tarea a ejecutar.

Los métodos *sched_proximo_indice()* y *sched_proxima_bandera()* aumentan *currTask* o *currFlag* al principio (por ello se los setea en -1 al inicializarlos). Luego, recorren sus respectivos arreglos (*tasks* y *flags*) hasta encontrar algún índice que no sea 0. En caso de que todos sean 0, el método devuelve 0. Una vez encontrado la próxima tarea/bandera a ejecutar, se invoca a una función de *screen.c* para que actualice en pantalla la tarea o bandera que se está ejecutando, y retorna el selector de segmento de la tarea/bandera a ejecutar.