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 se encarga de modificar una estructura del kernel distinta, los métodos que modifican a cada estructura se mantuvieron privados a su respectivo archivo; es decir, la única manera de modificar dichas estructuras es a través de los métodos de sus respectivos archivos.

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 del archivo y los métodos que se implementan en él. Por ejemplo, *screen.c* contiene todos los métodos para imprimir en pantalla, y ningún otro archivo posee métodos que impriman en pantalla.

Por otro lado, cada vez que se produce algún cambio en alguno de los *buffers* de pantalla (por ejemplo, al producirse una excepción o cuando una tarea ejecuta un *syscall*), se dibuja sobre la pantalla el *buffer* que haya sido modificado. De esta manera, se pueden apreciar automáticamente los cambios que se van produciendo. Sin embargo, ya que el *quantum* de cada tarea es de un *tick* de reloj, los cambios se producen muy rápidamente, impidiendo que se observen en pantalla. Por ello, dejamos incluido un *breakpoint* justo antes de realizar el salto a la próxima tarea, de manera que se aprecien los cambios en cada iteración.

Ya que las funciones bandera deben ejecutarse cada vez que se ejecuta la tarea bandera, antes de saltar a la respectiva tarea de la bandera, se posiciona el registro EIP de su TSS sobre el principio de la función bandera (el método que lo realiza se llama *void reiniciar_bandera(int flag)*, ubicado en *tss.c*). De esta manera, una vez que se terminó de ejecutar la función, se asegura que la próxima vez que sea ejecutada, la misma comience desde el principio y no al final.

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

    cmp byte [corriendoBandera], 0x00
    jne .pintar
    call desalojar_tarea_actual         ; una tarea llamo a la int 66
    call screen_modo_estado
    jmp .fin

    .pintar:
        push eax
        call pintar_buffer_bandera
        pop eax
        mov byte [corriendoBandera], 0x00
    .fin:
        popad
        jmp 0xB8:0x00       ; tarea_idle
        iret
```

Como se puede apreciar, se utiliza dicha variable global para analizar quién realizó la interrupción. En caso de que haya sido una bandera, se procede a dibujar el *buffer* devuelto por la misma sobre la pantalla.

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, direccion)*).

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, direccion, pagina)* de la clase *screen.c* para dibujarlas en el *buffer* del mapa.

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

También es necesario saber dónde se ubica la función bandera cuando se inicializan las TSS de las banderas, por lo que este archivo implementa un método que retorna la dirección física sobre la cual se ubica la función bandera:

```
unsigned int dir_funcion_bandera(int tarea){
    unsigned int *ptr_funcion = (unsigned int*) (dir_tareas[tarea] + 0x1FFC);

    return (unsigned int) (*ptr_funcion);
}
```

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 (y, por lo tanto, son invocados antes de que se empiecen a ejecutar las tareas).

El método *pintar_buffer_bandera(dir_buffer)* se encarga de dibujar donde corresponde lo devuelto por la función bandera.

En particular, *coordenadas(direccion)* se encarga de convertir una dirección de memoria en una posición de la pantalla. Luego, es utilizado por la función *asignar_dir(tarea, direccion, 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(direccion)* se encarga de dibujar a dónde fue dirigido el último misil y borra del mapa al anterior.

Por último, el método *borrar(tarea)* se encarga de eliminar del *buffer* de mapa todos los datos de la tarea pasada por parámetro, además de reemplazar la bandera de la tarea por una cruz (en el *buffer* de estado).

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 *mmu_inicializar_dir_tarea(tarea, dir)* descripto anteriormente, 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);
```

Además, contiene el método necesario para reinstaurar la ejecución de la función bandera a su inicio:

```
void reiniciar_bandera(int flag){
    tss_banderas[flag].eip = 0x40000000 + dir_funcion_bandera(flag);
}
```

1.8. sched.c

La implementación del *scheduler* contiene variables propias 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;
        borrar(currFlag);
    }
    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, en cuyo caso 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 su 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.