



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

System Programming - Tierra Pirata

Organización del Computador II
Primer Cuatrimestre de 2015

Grupo Diablo II / PC

Integrante	LU	Correo electrónico
Ciruelos Rodríguez, Gonzalo	063/14	gonzalo.ciruelos@gmail.com
Maddonni, Axel	200/14	axel.maddonni@gmail.com
Thibeault, Gabriel		



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>

Índice

1. Introducción	3
2. Desarrollo	4
2.1. Ejercicio 1	4
2.2. Ejercicio 2	5
2.3. Ejercicio 3	6
2.4. Ejercicio 4	7
2.5. Ejercicio 5	8
2.6. Ejercicio 6	8
2.7. Ejercicio 7	9
2.8. Ejercicio 8	10
3. Conclusiones	11

1. Introducción

Los sistemas operativos son el software que se ocupa de manejar y administrar los recursos del hardware y proveer servicios a los programas. Sin sistemas operativos sólidos, realizar la mayoría de las tareas que realizamos hoy con las computadoras serían imposibles.

En este trabajo nos proponemos aprender y entender como funcionan los mecanismos básicos que implementa un sistema operativo para manejar la memoria, manejar interrupciones, alternar tareas, etc.

A lo largo de la historia existieron muchos paradigmas de sistemas operativos, en general atados a las capacidades tecnológicas de la época. En este TP nos proponemos hacer un sistema operativo que trabaja en modo protegido de 32 bits, con paginación y multitarea. Teniendo esas características, nuestro sistema operativo *de juguete* es más avanzado en varios aspectos que el DOS original de Microsoft (que trabajaba en modo real y era monotarea), por lo que sentimos que va a ser un interesante desafío.

Para testear nuestro kernel durante el proceso de desarrollo utilizamos el software Bochs, un proyecto de código abierto, que permite emular una IBM PC tanto de 32 bits, como de 64 bits. También permite emular dispositivos y un BIOS, por lo cual es ideal para el desarrollo de sistemas operativos.

2. Desarrollo

2.1. Ejercicio 1

Inicialización de la GDT

Inicializamos la Tabla de Descriptores Globales con entradas para segmentos de código de nivel 0 y 3, otras para segmentos de datos de nivel 0 y 3, una para un segmento que describe el área de la pantalla de video, y la entrada correspondiente al segmento donde se guardará la tss de la tarea inicial. (Las entradas de gdt para las tss de las demás tareas son completadas al inicializarlas, como se explica en la sección correspondiente al Ejercicio 6).

Se utiliza desde el índice 8 por restricciones del trabajo práctico. Los segmentos de datos y códigos están organizados de tal forma que se superpongan direccionando los primeros 500MB de memoria (Sistema FLAT), utilizando bloques de 4K al setear el bit de granularidad en 1.

Los demás atributos fueron seteados de la siguiente manera:

Base y Límite: Como mencionamos anteriormente, los segmentos de código y datos están superpuestos. Comienzan en la dirección base 0x00000000, y el valor del límite 0x1F3FF corresponde la cantidad de bloques-1. Es decir, para cubrir 500MB se necesitan 128.000 bloques de 4K. El offset del último bloque es 127.999 (0x1F3FF en hexa). Con respecto al segmento de video, éste ocupa en memoria desde la posición 0xB8000 hasta la 0xC0000, es decir 32K de memoria, cuyo máximo offset o límite es el correspondiente al último byte (7999 = 0x7FFF). Para las tss, el límite es 0x68, pues miden 104 bytes cada una. Como base de la tarea inicial, seteamos la dirección 0x0000. ??????

Tipo: El tipo para los segmentos de código es 0x0A (executable, readable), mientras que para los de datos y video es 0x02 (readable, writable).

Sistema: El bit de system está seteado en 1 salvo en los segmentos correspondientes a las tss de las tareas, donde está activo bajo en 0 pues son potestad exclusiva del sistema operativo.

DPL: Los segmentos de datos y código de nivel 0 tienen DPL en 0x00, al igual que los segmentos de sistema y el de video, mientras que los de código y datos nivel 3 tienen DPL en 0x03.

Granularidad: El bit de G está activo sólo en los segmentos de datos y código ya que es necesario bloques de 4K para abarcar los 500MB.

P, L, D/B, AVL: Seteados en 1, 0, 1 y 0 respectivamente para todas las entradas.

Pasaje a Modo Protegido

Para pasar a modo protegido, completamos y cargamos la GDT usando la instrucción lgdt, que toma el descriptor de la GDT con el tamaño y la dirección de la misma, habilitamos A20 para habilitar el acceso a direcciones superiores a los 2²⁰ bits, seteamos el bit de PE del registro CR0, y saltamos a 0x40:modoprotegido donde el 0x40 corresponde al Índice del segmento de código de nivel 0 (índice 8 en la gdt), corrido 3 ceros (estos ceros son los del TI y RPL).

Código 1: Pasaje a modo protegido

```
; Habilitar A20
call habilitar_A20

; Cargar la GDT
lgdt [GDT_DESC]      ; cargo la estructura que esta en gdt.c

; Setear el bit PE del registro CR0
mov eax, cr0
or  eax, 1
mov cr0, eax

; Saltar a modo protegido
jmp 0x40:modoprotegido
```

Una vez trabajando en modo protegido, procedemos a establecer los selectores de segmentos de datos de nivel 0 (índice 9 en la gdt, corrido tres ceros correspondientes a los bits de TI y RPL), y el selector de segmento de video en fs (índice 12 en la gdt). Luego establecemos la base de la pila en la dirección 0x27000.

Código 2: Pasaje a modo protegido

```
modoprotegido:

; Establecer selectores de segmentos
xor eax, eax
mov ax, 1001000b
mov ds, ax
mov es, ax
mov gs, ax
mov ss, ax

mov ax, 1100000b
mov fs, ax

; Establecer la base de la pila
mov ebp, 0x27000
mov esp, 0x27000
```

Inicialización de la Pantalla

Para inicializar la pantalla llamamos a la función de screen.h *screen_inicializar*, que se encarga de pintar la pantalla con el mapa, las barras para los jugadores, e inicializar los slots vacíos y puntos en 0 de cada jugador, utilizando las funciones *screen_pintar_rect* para pintar un rectángulo de color, *print_dec* para imprimir los puntos, y *screen_pintar* para imprimir caracteres.

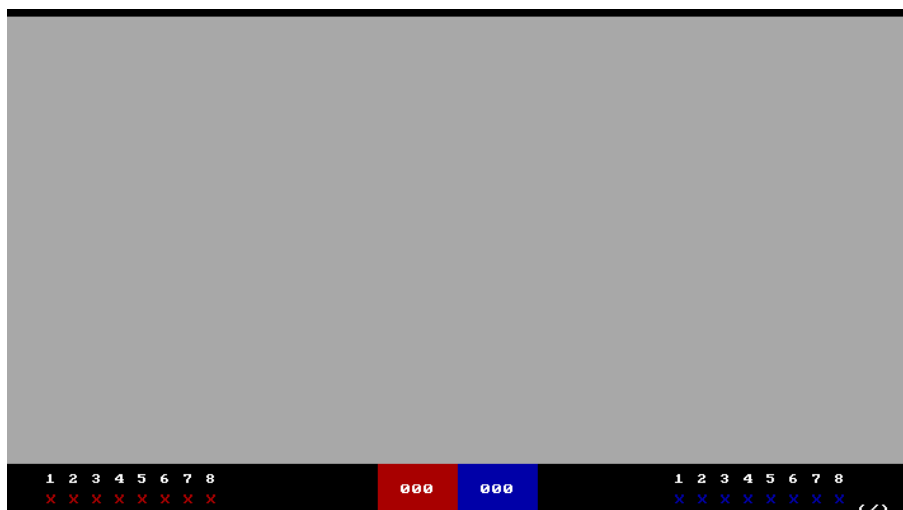


Figura 1: Pantalla Inicial.

2.2. Ejercicio 2

Inicialización de la IDT

Para inicializar la IDT se llama a una función en lenguaje C, "void idt_inicializar(void)". La IDT se representa mediante un arreglo (de tamaño 255) de *idt_entry*"(esta estructura fue definida como lo muestra el extracto de Código 3, según las especificaciones dadas en el manual de Intel). Esta función utiliza un

macro (que se encuentra en el extracto de Código 4) para inicializar cada una de las entradas necesarias de la IDT.

El macro define el offset como los 16 bits menos y más significativos (ya que éste se encuentra separado en dos campos de 16 bits cada uno) de la dirección de la tarea de atención de la interrupción correspondiente. El selector de segmento lo define como 0x0040, que es 0x8 (el índice del segmento de código de nivel de privilegio 0 en la GDT) shiftado 3 bits a la izquierda. Los atributos los define como 0x8e00, que representan un segmento presente (P = 1), un nivel de privilegio de 0 (DPL = 00), un tamaño de Gate de 32 bits (bits 8 a 12 de la Interrupt Gate = 0b01110), y los 7 bits restantes en 0.

La rutina de atención de cada interrupción se genera a partir de un macro (que se encuentra en el extracto de Código 5) que imprime el código de error correspondiente a pantalla y luego queda en un loop infinito.

La IDT ya inicializada se puede acceder tras ejecutar la instrucción `lidt[IDT_DESC]`. El descriptor de la IDT, `IDT_DESC`, es una estructura (definida como se ve en el extracto de código ??) que contiene el tamaño de la IDT y su dirección en memoria.

Código 3: Estructura de `idt_entry`

```
typedef struct str_idt_entry_fld {
    unsigned short offset_0_15;
    unsigned short segsel;
    unsigned short attr;
    unsigned short offset_16_31;
} __attribute__((__packed__, aligned (8))) idt_entry;
```

Código 4: Código del macro utilizado para inicializar la IDT

```
#define IDT_ENTRY(numero)
    idt[numero].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## numero)
        & (unsigned int) 0xFFFF);
    idt[numero].selsel = (unsigned short) 0x0040;
    idt[numero].attr = (unsigned short) 0x8e00;
    idt[numero].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## numero)
        >> 16 & (unsigned int) 0xFFFF);
```

Código 5: Código del macro utilizado para la rutina de atención de interrupciones

```
_isr%1:
    mov eax, %1
    push dword 0x0f0f
    push dword 0
    push dword 0
    push MENSAJE_ERROR_%1
    call print

    jmp \$
```

Código 6: Estructura de `IDT_Desc`

```
typedef struct str_idt_descriptor {
    unsigned short idt_length;
    unsigned int idt_addr;
} __attribute__((__packed__)) idt_descriptor;
```

2.3. Ejercicio 3

Para inicializar el directorio del kernel, lo que hacemos es, en la primera posición declarar la tabla de kernel (que será identity mapping), y luego ponemos todo el resto de directorio en 0 (bit de presente en 0, que indica que esas entradas no direccionan nada).

Luego inicializamos la tabla de kernel, que está en identity mapping, como dijimos anteriormente. Eso es bastante fácil, ya que podemos usar la misma variable para iterar y para decir la dirección física a la que direccionará una dirección virtual.

Para pintar la pantalla usamos las funciones que nos permiten pintar rectángulos, que no necesitan explicación dado que son muy simples.

2.4. Ejercicio 4

Inicialización de la MMU

Para administrar la memoria en el área libre contamos con un contador de páginas inicializado en la dirección 0x00100000. A medida que el sistema necesita una página, éste contador se incrementa en 4K, como muestra la siguiente implementación:

Codigo 7: Contador de Páginas Libres

```
void * siguiente_libre;

void inicializar_mmu()
{
    siguiente_libre = (void *) PAGE_COUNTER_INIT;
}

void * dar_siguiente()
{
    uint i;
    for(i = 0; i<1024; i++) ((pde *) siguiente_libre)[i].present = 0;
    siguiente_libre += 0x1000;
    return siguiente_libre - 0x1000;
}
```

Al crear una página, recorreremos todas entradas de la tabla seteando el bit de presente en 0 (sea ésta un directorio o tabla de páginas). Para simplificar la manipulación en el código de las pde y pte creamos dos estructuras en C correspondientes a las ya mencionadas:

Codigo 8: struct Page Directory Entry

```
typedef struct pde_t {
    unsigned char present:1;
    unsigned char read_write:1;
    unsigned char user_supervisor:1;
    unsigned char page_level_write_through:1;
    unsigned char page_level_cache_disable:1;
    unsigned char accessed:1;
    unsigned char reserved:1;
    unsigned char page_size:1;
    unsigned char global:1;
    unsigned char available_9_11:3;
    unsigned int base_address:20;
} __attribute__((__packed__, aligned (4))) pde;
```

Codigo 9: struct Page Table Entry

```
typedef struct pte_t {
    unsigned char present:1;
    unsigned char read_write:1;
    unsigned char user_supervisor:1;
    unsigned char page_level_write_through:1;
    unsigned char page_level_cache_disable:1;
    unsigned char accessed:1;
```

```

    unsigned char dirty:1;
    unsigned char page_table_attribute_index:1;
    unsigned char global:1;
    unsigned char available_9_11:3;
    unsigned int base_address:20;
} __attribute__((__packed__, aligned (4))) pte;

```

Inicialización de directorios y tablas para Tareas Pirata

Mapeo y Desmapeo de Páginas

Testeo de Paginación (item no implementado en la sol final)

2.5. Ejercicio 5

Interrupción de Reloj

Interrupción de Teclado

Interrupción de sistema (0x46)

2.6. Ejercicio 6

TSS

Los TSS (task state segment) son una parte vital en el manejo de tareas. Ellas se ocupan de guardar el información sobre una determinada tarea, más precisamente, el contexto de ejecución que tenía una determinada tarea cuando el procesador cortó su ejecución.

Entradas de la GDT

Las TSS, como todo segmento, tienen un descriptor que se declara en la TSS. En nuestro caso necesitaremos bastantes descriptores. Primero, uno para la tarea inicial (explicaremos más adelante que es esto), otro para la tarea idle, y luego 8 para cada jugador, es decir, en total declaramos 18 descriptores de TSS en la GDT.

La tarea inicial es un placeholder que debe llenarse antes de empezar a trabajar con tareas. Esto se debe a que cuando saltamos a nuestra primera tarea, el procesador intentará guardar el contexto de ejecución de la tarea que está corriendo actualmente, y si no declaramos un descriptor de segmento ad-hoc para esa tarea, todo va a explotar.

Veamos como deben ser inicializadas las TSS de las tareas.

La TSS de la tarea inicial puede ser inicializada con fruta, porque es un contexto que nunca vamos a retomar, como bien explicamos antes.

Para la tarea idle, es bastante straightforward todo, se debe inicializar todo en nivel kernel (la pila y los segmentos de código y datos son de kernel), además de poner el eip donde corresponde (al inicio del código de la tarea).

Para las tareas de los piratas, explicitamos el código a continuación (caso jugador A):

```

void tss_inicializar_tarea(uint indice_tarea, cual_t jugador, pde * cr3_nuevo)
{
    (...)

    tss_jugadorA[i].cs = 0x53;
    tss_jugadorA[i].ds = 0x5b;
    tss_jugadorA[i].es = 0x5b;
    tss_jugadorA[i].gs = 0x5b;
    tss_jugadorA[i].ss = 0x5b;
    tss_jugadorA[i].fs = 0x5b;

    tss_jugadorA[i].eflags = 0x202;
}

```



```

tss_jugadorA[i].iomap = 0xFFFF;

(...)

tss_jugadorA[i].esp = 0x0401000 - 12;
tss_jugadorA[i].ebp = 0x0401000 - 12;

tss_jugadorA[i].eip = 0x00400000;
tss_jugadorA[i].esp0 = (unsigned int) dar_siguiete() + 0x1000;
tss_jugadorA[i].ss0 = 0x48;

tss_jugadorA[i].cr3 = (uint) cr3_nuevo;
(...)
}

```

Veamos de donde salen todos esos números. El 0x53 y el 0x5b salen de que 10 y 11 son las entradas de la GDT de los segmentos de código y datos0 de nivel 3 respectivamente, entonces $10 \ll 3 | 0x3 = 0x53$, y similarmente para 0x5b.

Por otro lado, la pila se inicializa en $0x0401000 - 12$, dado que el final del código de la tarea va a estar en $0x0401000$, pero va a tener apilada 3 cosas, los dos parámetros y su dirección de retorno, a lo que se debe el -12. El EBP podría inicializarse en cualquier cosa, todo da lo mismo, dado que la tarea nunca va a retornar (por la misma razón la dirección de retorno puede ser cualquiera).

La ultima cosa importante a notar es que la pila de nivel 0 se inicializar en $\text{dar_siguiete}() + 0x1000$, porque cada tarea debe tener una pila de nivel 0 distinta, y ademas el esp debe apuntar al final de ese lugar (si no al pushear cosas va a pisar la página anterior y eso está mal).

Esta función se divide muchas funcionalidades con `mmu_inicializar_dir_pirata`, por lo tanto, para terminar de entender bien como es el proceso de inicialización de una tarea, deben entenderse bien ambas funciones, dado que son la parte más importante de este proceso.

2.7. Ejercicio 7

Estructuras del scheduler

La estructura en la que almacenamos los datos necesarios para la conmutación de tareas es muy simple. Básicamente son 3 datos

```

struct {
    uint indiceA;
    uint indiceB;

    cual_t proximo;
} sched_struct;

```

`indiceA` nos indica cual es el próximo índice en el que deberíamos comenzar a buscar una nueva tarea para ejecutar del jugador A (similarmente `indiceB`). Notar que este índice puede corresponderse con una tarea válida del jugador tanto como con una tarea muerta o inválida.

`proximo`, nos indica a que jugador le toca jugar, es decir, en que arreglo de tareas vamos a comenzar nuestra búsqueda.

Algoritmos del scheduler

La estructura que elegimos para el scheduler se ve fuertemente reflejada en los algoritmos. Primero notemos como inicializamos las estructuras, todas en 0, y elegimos, arbitrariamente, que el proximo jugador al que le tocará es el A (podríamos haber elegido obviamente cualquiera).

Ahora inspeccionamos la funcion `sched_proxima_a_ejecutar`. Lo que hace es muy simple. Si el jugador al que le toca jugar es el A, comenzara buscando por su arreglo, desde `indiceA`, algún pirata que este vivo, y en caso de encontrarlo hara 4 cosas: setear a B como el proximo jugador, setear la id de la

tarea actual con el id de la tarea seleccionada, inicializar mineros del jugador, si es que había mineros pendientes y finalmente devolver el índice en la gdt de la tss de la tarea seleccionada. En caso de no encontrarlo, hará exactamente lo mismo con el arreglo del jugador B, buscando piratas vivos y etc.

Interrupción de reloj

La última parte del scheduler que nos toca analizar es la parte que hace la conmutación de tareas, que es la interrupción de reloj.

La rutina de atención de interrupción de reloj es muy similar a la que nos dieron en clase. La única diferencia que tiene es que se fija si está activada la pantalla de debug, y en ese caso no hace nada (dado que no queremos que mientras la pantalla de debug esté activada, los piratas se sigan moviendo por el mapa).

El comportamiento del resto es realmente simple, llamamos a la función que nos da el índice de la gdt de la tss de la proxima tarea a ejecutar, la comparamos en el índice de la tarea actual (si la tarea es la misma no debemos saltar, porque saltaríamos a una tarea que tiene el bit de Busy en 1 y explota todo) y en caso de que sea distintos, saltamos.

Es importante notar que cuando se le reasigne la ejecución a una tarea, esta tarea va a volver a la linea que dice popfd y luego volverá a su ejecución común y corriente.

Modo Debug

El modo debug es fácil de hacer una vez que el resto de las cosas están bien hechas. Debimos agregar un par de variables globales que nos indiquen si el modo debug está activado y otra que nos indique si se está mostrando la pantalla de debug en un momento dado.

Cuando nos llega una interrupción de las primeras 20, lo que hacemos es guardar toda la información que este disponible en ese momento (la que debemos guardar) y luego llamar a `game_pirata_exploto`.

En `game_pirata_exploto` (ademas de hacer las limpiezas de estructuras que correspondan) lo que hacemos es chequ si está el modo debug activado o no. En caso afirmativo, cargamos ciertas cosas que sean necesarias y llamamos a la función `screen_debug` que es realmente simple, muestra todos los datos en pantalla, mientras siga activado el modo debug. Cuando se vuelve a apretar 'y', el modo debug se desactiva y se sale de el loop, restaurando la pantalla como estaba (que previamente había sido backupeada).

2.8. Ejercicio 8

3. Conclusiones

aa