Organización del Computador II

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

TP N°3

Grupo Pistacho/Chungo

Integrante	LU	Correo electrónico
Axel Straminsky	769/11	axelstraminsky@gmail.com
Lucas Romero	440/12	lucasrafael.romero@gmail.com
Matias Pizzagalli	257/12	matipizza@gmail.com

1. GDT y modo protegido.

Para empezar el primer objetivo del kernel es pasar a modo protegido. Para realizar esto primero deshabilita las interrupciones, habilita la línea A20 y carga la Global Descriptor Table (GDT). El kernel carga la GDT con la instrucción "lgdt [GDT_DESC] "donde GDT_DESC es un descriptor en un struct declarado en el archivo gdt.h. El descriptor contiene el tamaño de la GDT y su dirección (En ese orden).

La GDT que carga es inicializada en el archivo gdt.c. En principio se cargan 4 descriptores: 2 segmentos de datos y 2 de código. De esos segmentos uno de código y uno de datos tienen privilegio 0 (kernel) y los otros dos cuentan con privilegio 3 (usuario). Estos cuatro descriptores se encuentran en los índices 0x12 a 0x14 ya que las primeras 18 entradas estan reservadas por la cátedra.

Los cuatro segmentos tienen base en 0x00000000 y límite 0x6FFFF que corresponden con un modelo de segmentación flat. Además, los segmentos de código permiten la lectura y los de datos permiten la escritura. Todos los descriptores tienen el bit Default/Big seteado en 1 de manera que los segmentos son de 32 bits y están todos presentes. Para los 4 segmentos, el bit de granularidad está en 1.

Entonces, los 4 descriptores de segmento quedan seteados de esta manera:

Atribute	Code Kernel	Data Kernel	Code User	Data User
limit(0:15)	0xFFFF	0xFFFF	0xFFFF	0xFFFF
base(0:15)	0x0000	0x0000	0x0000	0x0000
base(23:16)	0x00	0x00	0x00	0x00
type	0x0A	0x02	0x0A	0x02
s	0x01	0x01	0x01	0x01
dpl	0x00	0x00	0x03	0x03
p	0x01	0x01	0x01	0x01
limit(16:19)	0x06	0x06	0x06	0x06
avl	0x00	0x00	0x00	0x00
l	0x00	0x00	0x00	0x00
d/b	0x01	0x01	0x01	0x01
g	0x01	0x01	0x01	0x01
base(31:24)	0x00	0x00	0x00	0x00

Ya con la GDT inicializada y cargada podemos entrar en modo protegido. para hacer esto simplemente cargamos el contenido del registro de control 0 (cr0) en eax, efectuamos un 'or eax, 1' y movemos ese resultado de vuelta a cr0. De esta manera queda seteado el bit PE (Protect Enable) y empezamos a trabajar en modo protegido.

Lo siguiente sería darle a los selectores de segmento su valor correspondiente. Primero efectuamos la instrucción "jmp 0x90:mp ", (mp es un label situado exactamente luego de la instrucción) con esta simple instrucción saltamos al segmento de código de privilegio 0 y ponemos en CS el índice para este segmento. Para el resto de los selectores simplemente movemos 0x98 al selector (índice 0x13) para que queden apuntando al segmento de datos del kernel. Luego de realizar esto seteamos la pila del kernel con la instrucción "mov esp, 0x27000 ".

Lo que se nos pide a continuación es armar otro segmento que describa la posición en memoria de la pantalla que solo pueda ser accedido por el kernel. Con este fin declaramos en el archivo gdt.c otra entrada a la gdt en el índice 0x15 que describa un segmento de datos de 32 bit con privilegio 0 y que sea de lectura y escritura. La base de este nuevo segmento está en la dirección 0xB8000 y tiene un límite de 0x7FFF. El bit de granularidad está en 0 con lo cual el límite se calcula normalmente.

Atribute	Screen Area
limit(0:15)	0x7FFF
base(0:15)	0x8000
base(23:16)	0x0B
type	0x0A
s	0x01
dpl	0x00
p	0x01
limit(16:19)	0x00
avl	0x00
l	0x00
d/b	0x01
g	0x00
base(31:24)	0x00

Con este último segmento podemos terminar el primer ejercicio y escribir la rutina requerida. Esta rutina se puede dividir en 2 partes. La primera, una llamada a una función en C que limpia la pantalla y la deja en blanco, esta función se encuentra en screen.c. La segunda, son dos pequeños ciclos en Assembler que cada uno se encarga de pintar la primera o la última línea de la pantalla de fondo negro con caracteres blancos usando el segmento descripto en el punto anterior.

2. IDT y rutinas de atención de excepciones

Ya en modo protegido el objetivo consiste en armar una Interruption Descriptor Table (IDT) y escribir las rutinas de atención, para poder atender en principio las excepciones que puedan surgir y más adelante las interrupciones de reloj, teclado y syscalls.

Para este propósito hay definida una macro en C para completar las entradas de la IDT.

```
#define IDT_ENTRY(numero, privilegio)
idt[numero].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## numero) & (unsigned int) 0xFFFF);
idt[numero].segsel = (unsigned short) 0x90;
idt[numero].attr = (unsigned short) (0x8E00  | (privilegio << 13));
idt[numero].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## numero) >> 16 & (unsigned int) 0xFFFF);
```

De esta forma, cuando llamamos a idt_inicializar() inicializamos las primeras 20 entradas de la IDT con los siguientes valores:

Atributo	Valor
Selector	0x90 (Kernel data segment)
Offset	(Offset correspondiente a la isr de la excepción)
Tipo	0x0E (Interrupt gate, 32-bit)
DPL	0x00
Presente	0x01
Sin usar	0x00

Para las rutinas de atención de las excepciones escribimos una macro en assembler que se ocupa de detectar la excepción, imprimirla en pantalla y colgar la ejecución ejecutando jmp $.

```
%macro ISR 1
global _isr%1

_isr%1:
.loopear:

    cli
    pushad
```

```
    mov eax, interrupciones
    mov ebx, %1

    mov eax, [eax + ebx * 4]

    push eax
    call imprimir_interrupcion
    pop eax

    popad
    sti


%    jmp $

%endmacro
```

Donde interrupciones es un arreglo de strings que contienen los mensajes a imprimir por cada excepción e imprimir_interrucion es una rutina que toma un string y lo imprime en la parte correspondiente al último problema.

Para testear esto generamos excepciones (tanto intencionales como accidentales) como divide by zero, page fault, general protection, invalid tss; entre otras. Los resultados fueron los esperados.

3. Paginación y rutinas de pantalla

Lo primero fue escribir las rutinas que escriben los buffer de video para la pantalla de mapa y de estado. Estas funciones fueron hechas en C y escriben dos buffer, uno en la dirección 0x2D000 y otro en la dirección 0x2E000. Estos corresponderán a las pantallas de estado y de mapa respectivamente. Están escritas para presentar una versión genérica de las figuras 9 y 10.

Luego, para activar paginación lo único que hay que hacer es setear el bit 31 del registro cr0, pero para llegar a eso primero hay que armar las estructuras necesarias. En este caso, un directorio de páginas y por lo menos una entrada en este directorio con una tabla de páginas que tenga entradas válidas para mapear memoria.

Con este fin, cargamos en el registro cr3 la posición que ocupará el page directory del kernel (0x28000) y llamamos a la función mmu_inicializar_dir_kernel. Es una función escrita en C y que, como lo indica su nombre, se encarga de inicializar el mapa de memoria del kernel.

Previamente a esto, utilizamos las funciones inicializar_page_directory e inicializar_page_table, las cuales ponen en 0 todos los atributos de las entradas del page directory y page table, salvo el atributo Read/Write que dejan en 1. Esto se hace tanto para la página del page directory como para las 2 page tables. Ya inicializadas las tablas, se las modifica para que realicen identity mapping, las 1024 entradas de la primera tabla (los primeros 4 MB) y las primeras 896 entradas de la segunda tabla (los 3,5 MB restantes). Una vez que tienen seteados los valores, son finalmente agregadas al page directory. Las primeras 256 páginas (1 MB) las mapea con privilegio 0 y el resto con privilegio 3

Finalmente el directorio, y tablas quedan con los siguientes valores:

PDE	Base	G	PS	A	PCD	PWT	U/S	R/W	P
0	(Base de la tabla)	0b	0b	0b	0b	0b	0b	1b	1b
1	(Base de la tabla)	0b	0b	0b	0b	0b	0b	1b	1b
2 a 1023	0x00	0b	0b	0b	0b	0b	0b	1b	0b

Tabla	PTE	Base	G	PS	A	PCD	PWT	U/S	R/W	P
0	(primeras 256 n)	n	0b	0b	0b	0b	0b	0b	1b	1b
0	(restantes 768 n)	n	0b	0b	0b	0b	0b	1b	1b	1b
1	(primeras 896 n)	n + 1024	0b	0b	0b	0b	0b	1b	1b	1b
1	(restantes)	0x00	0b	0b	0b	0b	0b	0b	1b	0b

4. Memory Management Unit

Para inicializar el mapa de memoria de las tareas, usamos dos funciones: inicializar_tareas y mmu_inicializar_dir_tareas. El objetivo de la primera es inicializar los directorios y tablas de páginas de las tareas. Primero inicializa todos los valores en 0, salvando el bit de read/write, luego mapea los primeros 7,5 MB de páginas con identity mapping de nivel 0 y agrega esas 2 páginas en las primeras 2 posiciones del page directory. Finalmente termina agregando las 3 páginas de la tarea en la posición 0x100 del directorio de páginas.

Cada tarea cuenta con 6 páginas: 1 para el page directory, 3 para las page tables, 1 para la pila de código, y otra para la pila de la bandera. Estas páginas se encuentran a partir de la posicion 0x31000. La función mmu_inicializar_dir_tareas se encarga de mapear dos páginas en el mar y una página en la tierra en el mapa de memoria de cada tarea. Luego de esto copia las dos páginas de código de la tarea y la bandera en el area de mar y setea los atributos relacionados con la memoria para cada tarea. Finalmente, inicializa el arreglo global direcciones_tareas, el cual guarda las direcciones físicas actuales de las 3 páginas de cada tarea, información que utilizamos en las funciones de pantalla.

Al terminar la inicialización de la mmu cada tarea mapea las siguientes direcciones:

Dirección virtual	Dirección física
0x00000000-0x0077ffff	0x00000000-0x0077ffff
0x40000000-0x40001fff	0x100000 + (0x2000 * id_tarea) - 0x101fff + (0x2000 * id_tarea)
0x40002000-0x40002fff	0x00000000-0x00000fff

Para realizar las dos funciones descriptas mas arriba nos proponian implementar y utilizar dos funciones mmu_mapear_pagina y mmu_unmapear_pagina, pero por decisiones nuestras, decidimos implementar la primera tal cual proponen pero la segunda retocarla para darle un mejor uso. Ambas funciones luego nos resultan utiles para otras aplicaciones como en los syscalls de navegar y fondear.

Lo que hace la primera de ellas basicamente es obtener la informacion sobre la page table y la page directory de la direccion virtual pasada por parametro, para acceder a las entradas en el directorio y tabla de paginas correspondiente. para mapear ahi la direccion fisica que nos pasan por parametro con los atributos tambien pasados por parametro.

```
void mmu_mapear_pagina(unsigned int virtual, unsigned int cr3, unsigned int fisica, unsigned int attr) {

        page_directory_entry* page_directory = (page_directory_entry*) cr3;
        virtual = virtual >> 12; //borro el offset
        unsigned int pte = virtual & 0x3ff;
        virtual = virtual >> 10; //borro el table
        unsigned int pde = virtual;

        unsigned int page_table_address = (page_directory[pde].page_table_address << 12);

        page_table_entry* page_table = (page_table_entry*) page_table_address;

        page_table[pte].present = attr & 0x1;
        page_table[pte].read_write = (attr & 0x2) >> 1;
        page_table[pte].user_supervisor = (attr & 0x4) >> 2;
        page_table[pte].physical_page_address = (fisica >> 12);

}
```

La segunda funcion, mmu_unmapear_pagina, decidimos implementarla diferente para poder utilizarla de mejor manera. Lo que decidimos es que la funcion en vez de unmapear la direccion virtual pasada por parametro, que solamente nos devuelva en que direccion fisica se encuentra mapeada, para luego nosotros encargarnos de pisar esa misma direccion con la funcion descripta mas arriba.

La implementacion de la funcion basicamente lo que hace es descomponer la direccion virtual en sus tres componentes para luego utilizar esas componentes para acceder a las entradas en el directorio y tabla de paginas correspondientes y obtener la direccion fisica a donde se encuentra mapeada esa direccion virtual.

```
unsigned int mmu_unmapear_pagina(unsigned int virtual, unsigned int cr3) {

        page_directory_entry* page_directory = (page_directory_entry*) cr3;
        virtual = virtual >> 12; //borro el offset
        unsigned int pte = virtual & 0x3ff;
```

```
        virtual = virtual >> 10; //borro el table
        unsigned int pde = virtual;

        page_table_entry* page_table = (page_table_entry*) (page_directory[pde].page_table_address << 12);
        unsigned int fisica = (page_table[pte].physical_page_address << 12);

        return fisica;

}
```

Adicionalmente, aprovechamos para obtener los offsets de las funciones bandera. Para obtener estos offsets de la dirección virtual 0x40001FFC de cada tarea, lo que hicimos fue lo siguiente: como la dirección virtual 0x40001FFC hace referencia a la segunda página física de cada tarea, con un offset de 0x1FFC, creamos un arreglo global llamado offsets_funciones_bandera, en donde recopilamos esta información. Para esto, directamente accedimos a las posiciones 0x10000, 0x12000, etc., con un offset de 0x1FFC. Luego, a la hora de inicializar o reinicializar las TSSs de las banderas, utilizamos directamente los offsets de este arreglo, a los cuales les sumamos la dirección virtual 0x40000000, valor que utilizamos como EIP de las mismas.

5. Interrupciones de hardware y syscalls

Al igual que para las excepciones, usamos el define en el el archivo idt.c para inicializar las entradas en la IDT que corresponderán a las interrupciones de reloj (Int 0x20), teclado (Int 0x21) y syscalls (Int 0x50 e Int 0x66). Las entradas de la IDT quedan con la siguiente información:

Atributo	Int 0x20 y 0x21	Int 0x50 y 0x66
Selector	0x90 (Kernel data segment)	0x90 (Kernel data segment)
Offset	(Offset correspondiente a la isr de la interrupción)	(Offset correspondiente a la isr de la interrupción)
Tipo	0x0E (Interrupt gate, 32-bit)	0x0E (Interrupt gate, 32-bit)
DPL	0x00	0x03
Presente	0x01	0x01
Sin usar	0x00	0x00

Con las entradas de la IDT ya definidas, el paso siguiente es escribir las rutinas de atención. En esta etapa, solo se le dan funcionalidades básicas, sin embargo posteriormente las interrupciones serán las que realicen el trabajo de scheduling.

Inicialmente, la interrupción de reloj solo se encarga de llamar a la función proximo_reloj. La interrupción de teclado muestra los buffer de video de mapa y estado e imprime un número en la esquina superior derecha de la pantalla. Las interrupciones 0x50 y 0x66 cambian el valor de eax por 0x42

Estas son las rutinas:

```
_isr32:
    cli
    pushad

    call fin_intr_pic1

    call proximo_reloj

    popad
    sti
    iret

_isr33:

    cli
    pushad
    call fin_intr_pic1
```

```
    xor eax, eax
    in al, puerto_teclado

    push eax
    call interrupcion_teclado
    add esp, 4

;Interrupcion_teclado es una funcion en C que imprime en pantalla el numero
;presionado o que cambia los buffer a mapa o estado segun corresponda

    popad
    sti
    iret

;isr 0x50
_isr80:

    mov eax, 0x42
iret

;isr 0x66
_isr102:

    mov eax, 0x42
iret
```

6. TSS

Lo primero que hicimos fue definir las entradas de la GDT que corresponderían a los descriptores de las TSS. Estas ocupan desde el índice 23 al 40. Son 2 por cada tarea (navío y bandera), una para la tarea idle y una para la tarea inicial. Todas son cargadas con la misma información. El atributo base es luego cargado dinámicamente al inicializar las tss. Las entradas de la GDT quedan con la siguiente información para todas las tareas.

Atributo	Valor
limit(0:15)	0x0067
base(0:15)	0x0000
base(23:16)	0x00
type	0x09
s	0x00
dpl	0x00
p	0x01
limit(16:19)	0x00
avl	0x00
l	0x00
d/b	0x00
g	0x00
base(31:24)	0x00

Con las entradas de la GDT inicializadas, el paso siguiente es setear los valores del tss para cada tarea, incluyendo las tareas idle e inicial. El enuniciado dicta que valores deben llevar las tss de las tareas idle, navío y bandera. En el caso de la tss inicial solo hay que poner una tss valida en la gdt pues solo se usa para dar el primer salto. Para inicializar la tarea idle, llamamos a la función tss_inicilizar_tarea_idle

Atributo	idle
cs	0x90
ds, es, fs, gs, ss	0x98
registros de propósito general	0x00
ebp, esp	0x40001C00
eflags	0x202
eip	0x40000000
esp0	0x2A000
ss0	0x98
cr3	0x28000

Una de las particularidades del procesador es que permite una conmutación de tareas automática, facilitando bastante el trabajo. Usando la instrucción `jmp selector: offset` , donde selector es un selector de segmento que apunte a una entrada valida de tss en la gdt, el procesador ejecuta un task switch hacia esa tarea sin la necesidad de trabajo adicional.

7. Scheduler

Para inicializar los arreglos de tss de las tareas usamos la funcion tss_inicializar. Es una función en el archivo tss.c que se ocupa de inicializar las tss de las tareas y las banderas. Los valores son los provistos por la cátedra y quedan de la siguiente manera:

Atributo	navío	bandera
cs	0xA3	0xA3
ds, es, fs, gs, ss	0xAB	0xAB
registros de propósito general	0x00	0x00
ebp, esp	0x40001C00	0x40001FFC
eflags	0x202	0x202
eip	0x40000000	[0x40001FFC] + 0x40000000
esp0	0x35000 + (0x6000 * id_tarea)	0x36000 + (0x6000 * id_tarea)
ss0	0x98	0x98
cr3	0x31000 + (0x6000 * id_tarea)	0x31000 + (0x6000 * id_tarea)

Donde id_tarea es el número de tarea a la cual se hace referencia.Los campos no mencionados tienen valor 0x00.

El paso siguiente es implementar las funciones sched_proximo_indice() y sched_proxima_bandera. Estas dos funciones las implementamos en C en el archivo sched.c. Y las escribimos así:

```
unsigned short sched_proximo_indice() {

        unsigned short i = (ultima_tarea_ejecutada) % 8;

    while (tareas_corriendo[i] == 0) {

        i = (i + 1) % 8; //Si no hay tareas corriendo, se cuelga.

    }

    return tareas_corriendo[i];

}
```

ultima_tarea_ejecutada es una variable global sobre la que se escribe el id_tarea de la última tarea que se ejecutó y el array tareas_corriendo es un array que tiene en cada índice el índice de la gdt correspondiente a la tarea si la tarea está corriendo o 0x00 si la tarea fue desalojada.

```
short sched_proxima_bandera() {
```

```
        static unsigned short i = 0;
        while(i < 8) {

                if (banderas_corriendo[i] != 0) {
                        i++;
                        return banderas_corriendo[i-1];

                } else {

                        i++;
                }
        }

        i = 0;
        return -1;

}
```

Donde el array banderas_corriendo es un array que tiene en cada índice el índice de la gdt correspondiente a la bandera si la tarea está corriendo o 0x00 si la tarea fue desalojada.

Con esas dos funciones hechas, lo siguiente que pide el enunciado es modificar la int 0x50 para que responda a los syscalls según corresponda. Lo primero que hace la int 0x50 es desactivar las interrupciones y chequear que quien la llamó no haya sido una bandera. Si ese fuera el caso, desaloja la tarea y la bandera, e imprime en pantalla el último problema. Si la llamó una tarea compara el parámetro que se le pasa por eax y decide que servicio llamar: game_fondear, game_canonear o game_navegar. Estas son 3 funciones escritas en C en el archivo game.c que realizan el servicio en sí. Si el parametro en eax no coincide con ningún código de servicio, salta al final de la interrupción. Al final la int 0x50 vuelve a setear las interrupciones y salta a la tarea idle.

A continuación modificamos la int 0x20 (reloj) para que en cada tick haga la conmutación de tareas. Primero chequea que no se esté ejecutando una bandera, de ser así desaloja la tarea y su bandera. Después chequea que la próxima tarea a saltar no sea la misma que se está ejecutando, si es así no salta y simplemente hace un iret. Si el índice al cuál saltar no corresponde a la tarea actual, escribe el selector en una variable local y hace un jmp far usando esa variable.

Una vez hecho esto, agregamos un contador: contador_bandera en 0. Cada vez que se va a saltar a una tarea, se aumenta en 1 el contador. Si el contador está en 3 empieza a llamar, una por una las banderas hábiles usando la función sched_proxima_bandera. Una vez que llamó a la última vuelve a setear el contador en 0 para que la próxima interrupción de reloj llame a la tarea que corresponda.

Finalmente, lo único que queda es modificar las isr de las excepciones para que impriman en pantalla el último problema, desalojen la tarea que las produjo y salten a la idle. Para hacer esto lo primero que chequea la excepción es si tiene un error code o no, porque esto cambia la forma de pushear parámetros para la función imprimir_ultimo_problema, una función escrita en C que se encuentra en el archivo screen.c. Ya llamada la función chequea si la produjo una bandera o una tarea, y desaloja ambas de acuerdo al resultado. Terminado esto hace un jmp a la tarea idle.