DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

Organización del Computador 2
2do Cuatrimestre - 2017

Grupo: Itanium 2

Integrante	LU	Correo electrónico
Lucas De Bortoli	736/15	lu_cas_.97@hotmail.com.ar
Luciano Martín Galli	534/15	lucianogalli@outlook.com
Lautaro Perez Lopez	618/16	lautaroperezlopez@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

Índice

1. Ejercicios

1.1. Ejercicio 1

1.1.a. Tabla de Descriptores Globales (GDT)

En gdt.c seteamos las entradas de la GDT, donde se describen los segementos. La primer entrada se completa con ceros, y las 6 siguientes entradas no se completan por la consigna del ejercicio. Desde la posición 8 en adelante, se completan dos para código y datos de nivel de privilegio 0, y otras dos para nivel 3. Las cuatro mapean en los primeros 500MB de memoria.

1.1.b. Pasar a modo protegido

Para realizar el pasaje a modo protegido, en kernel.asm realizamos:

- Deshabilitmos interrupciones, con la instrucción *cli*.
- Cargamos la GDT usando *lgdt* sobre la dirección de la misma.
- Encedemos el bit PE del registro CR0.
- Jump a la siguiente siguiente instrucción (usamos como selector 0x50 y la etiqueta modoprotegido de offset).

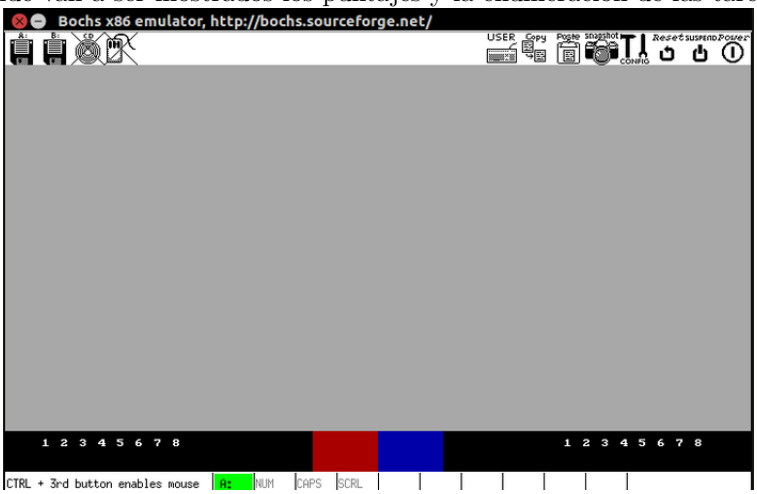
Una vez que estamos en modo protegido, cargamos los registros de segmento. Seteamos también la base y el tope de la pila en la dirección 0x27000

1.1.c. Segmento adicional

En gdt.c se agrega una nueva entrada a la GDT para asignar un segmento de memoria para el video, con el privilegio correspondiente para que sólo pueda ser utilizado por el kernel.

1.1.d. Limpiar pantalla

Implementamos en screen.c la función *screen_limpiar* para limpiar la pantalla, y otras funciones para pintar el fondo de gris, los rectángulos rojo y azul donde van a ser mostrados los puntajes y la enumeración de las tareas.



1.2. Ejercicio 2

1.2.a. IDT

En idt.c agregamos las entradas de la idt, que son las interrupciones que va a reconocer nuestro sistema. Hay que agregar en este punto las 20 interrupciones del sistema (de la 0 a la 19). En isr.asm se implementan los handlers para las interrupciones, es decir, como se van a resolver las mismas una vez que se produzcan. Utilizamos la macro ISR brindada por la cátedra para esto, agregando los mensajes correspondientes para cada interrupción.

1.2.b. Test de interrupción

En kernel.asm se debe inicializar y cargar la IDT. En ese momento las interrupciones del procesador ya empiezan a funcionar.Probamos el correcto funcionamiento de las interrupción intentando hacer una división por cero:

```
mov edx,0
mov ecx,0
mov eax,5
div ecx
```

Como era de esperar, se imprime en pantalla el mensaje $\#DE : Error\ de\ division$

1.3. Ejercicio 3

1.3.a.

Este ej es similar al 1.d

1.3.b. Directorio y tabla de páginas para el kernel

La función *mmu_inicializar_dir_kernel* llama a dos funciónes auxiliares.

- mmu_inicializar_page_dir: Inicializa el directorio de páginas, en la dirección 0x27000, con todas las entradas en 0, menos la primera, con los bits *present* y *readwrite* en 1, y carga la base con los 8 bits más significativos de la dirección de la tabla de páginas, 0x28000.
- mmu_inicializar_page_table: Inicializa la tabla de páginas en la dirección 0x28000. Para cada entrada se setean los bits *present* y *readwrite* en 1, y la base de cada entrada coincide con el índice de la misma.

Con esta configuración se permite mapear mediante identity mapping, las direcciones de la 0x00000000 a la 0x003FFFFF. Para cualquier dirección lineal en ese rango, los 10 primeros bits estarán en 0, por lo que entra siempre por la primer entrada del *page directory*. Entre los bits 21 y 12, el rango siempre se mantiene entre 0 y 767, lo que está en el rango de la *page table*. Como la base de cada entrada coincide con su índice, entonces la dirección física al sumarse el offset (los 12 últimos bits), quedará equivalente a la lineal en el rango pedido.

1.3.c. Activar paginación

En kernel.asm hacemos *call mmu_inicializar_dir_kernel*

1.3.d. Nombre grupo

Se utiliza la función *screen_pintar_name*, que muestra en pantalla el nombre del grupo alineado a la derecha, con color de fuente negro y fondo del mismo que haya en esa posición.

1.4. Ejercicio 4

1.4.a. Inicializar mmu

Para el manejo de memoria mediante paginación usamos un contador de páginas libres. La función *dame_pagina_libre* devuelve una nueva dirección de memoria para utilizar, cada vez que se invoca se le suma a una variable global *proxima_pagina_libre* el valor 0x1000, que es igual al tamaño de página. Esta variable se inicializa en la función *mmu_inicializar* con un valor de 0x10000, esta es una posición válida de memoria libre.

1.4.b. mmu_inicializar_dir_pirata

Esta función se va a encargar de inicializar los page directory y page table de un pirata, copiar su código a la posición inicial, y mapearle todas las posiciones recorridas por otros piratas. Recibe como parámetros un puntero al jugador al que pertence, un booleano que determina si es minero o explorador, y una coordenada del mapa que va a ser su posición inicial. En la función utilizamos *mmu_inicializar_page_dir* para crear un page directory para la tarea en una página libre. Luego, cargando en el *cr*3 la dirección del pd recién creado, se mapea la dirección física correspondiente al puerto del pirata con la dirección 0x400000, y se copia el código que le toca, dependiendo si es explorador o minero. Al final, se le mapea a la tarea las posiciones que los otros piratas descubrieron previamente, para poder ir a buscar un botín en el caso de ser minero, y no volver a mapearlas si es un explorador.

1.4.c. Funciones para mapear y desmapear

- mmu_mapear_pagina: le agregamos los parámetros "rw" y "us", para decidir al momento de llamar a la función si las páginas mapeadas permitirán su escritura y/o ser accedidas a nivel usuario. La función se encarga, dado un puntero a un page directory y la direcciones fisica y virtual a ser mapeadas, completar los campos del primero y completar una page table correspondiente

- mmu_unmappear_pagina: pone en 0 el bit de presente para la dirección que se desea desmapear.

1.5. Ejercicio 5

1.5.a. Completar la IDT

Se completa en la IDT las siguientes entradas: reloj = IDT_ENTRY_INT(32), teclado = IDT_ENTRY_INT(33), software = IDT_ENTRY_INT(0x46).

1.5.b. Rutina de interrupción de reloj

La rutina es *_isr*32: Al comienzo se guardan los registros con la instruccion pushad, a continuación se llama a la función fin_intr_pic1 para comunicarle al pic que se atendió la interrupcion. Luego se invoca a *screen_actualizar_reloj* que sirve para actualizar la figura del reloj en la pantalla. Esta misma interrupción se encarga de hacer el salto a la proxima tarea a ser ejecutada, de ser necesario (si no hay piratas activos se mantiene en la idle). Consigue el selector de segmento de la tarea correspondiente con la funcion *sched_tick*. Por último se restauran los registros y se retorna de la interrupcion con *iret*.

1.5.c. Rutina de interrupción de teclado

La rutina es *_isr*33: Luego de preservar registros y comunicar al pic que se antendió la interrupción, se lee la entrada de teclado con *in al*, *puerto_teclado*. A partir de aquí se toman decisiones dependiendo de la tecla presionada. En este sistema, *puerto_teclado* = 0x60. Por último se restauran los registros y se retorna de la interrupcion con *iret*.

1.5.d. Rutina de interrupción de software

Esta interrupción, al ser de software, va a saltar únicamente cuando sea llamada por las tareas, mediante la instrucción *int* seguida por el número de la interrupción, 0x46. Para este punto lo único que hace es mover el valor 0x42 al registro *eax*.

1.6. Ejercicio 6

1.6.a. Completar GDT con TSS inicial e idle

Se completa la GDT con dos nuevas entradas para la tss inicial y la tss idle. Estas tienen el bit de granularidad en cero. Usamos como límite 0x67, que es el mínimo valor que puede tener ya que el tamaño de un segmento tss debe ser mayor o igual a 0x68. Además se setean el bit de presente y el de db en uno por lo que el descriptor queda como válido y en 32bits. Por último se pone el tipo en 0x09 que es el correspondiente a un descriptor de tss en 32bits.

1.6.b. Rellenar la TSS de la tarea idle

En la funcion *tss_inicializar*, se asigna en la base de la entrada de la gdt correspondiente la dirección a una variable global llamada *tss_inicial*, y se llama a la función *tss_inicializar_idle*. Allí se modifica la base de la entrada a la gdt correspondiente para que apunte a la variable global *tss_idle*. Luego se completa la información de la tss asignandole sus respectivos segmentos, el valor de eip apuntando a 0x16000, y como se utiliza la pila del kernel, tanto esp como ebp

van a tener asignados la dirección 0x27000. Los eflags, al tener las interrupciones habilitadas, se setean en 0x202, el valor de iomap en 0xFFFF y los registros de propósito general en cero.

1.6.c. Rellenar TSS libre

Tenemos la función *tss_inicializar_tarea_pirata* que se encarga de completar los campos de la tss de una nueva tarea a ser creada, según el índice de la misma, recibido por parámetro. Todas las tss ya están creadas en un arreglo para cada jugador, llamados tss_jugadorA/tss_jugadorB, cuyo tamaño es la máxima cantidad de piratas posibles a lanzar para cada uno (8). Se completa en la entrada correspondiente de la gdt los datos para la tarea, y luego se rellena la tss propiamente dicha. En el campo de eip se pone la dirección 0x400000, donde fue mapeado el inicio del código de las tareas. El cr3 se consigue invocando a la función *mmu_inicializar_dir_pirata* , y el iomap es de 0xFFFF. Para la pila de la tarea se le asigna el final de la página donde comienza el código de la tarea, menos 12 bytes porque las tareas pueden recibir argumentos por la pila, como por ejemplo la posición de un botín para un minero. Los eflags se setean en 0x202, y los segmentos reciben su valor correspondiente acorde a los índices de la gdt. Para la pila de nivel cero, se pide una nueva página libre en el área libre del kernel, y se la sitúa al final de la misma. Los registros de propósito general comienzan en cero. Además se setea como activa la tarea en la estructura que utiliza el scheduler, $tareas\_A/tareas\_B$, según cual sea el jugador.

1.6.d. Entrada en la GDT para la tarea inicial

Se crea en la gdt la entrada para la tarea inicial, que será la primera en ser cargada.

1.6.e. Entrada en la GDT para la tarea idle

Lo mismo para la idle. Como es una tarea que se debe correr a nivel kernel, el dpl es 0.

1.6.f. Ejecutar la tarea idle

Primero cargamos en el task register a tarea inicial, luego se salta a la tarea idle moviento a la variable *selector* definida previamente, el valor 0x70; y ejecutando la instrucción $jmp\ far\ [offset]$, siendo esta otra variable ya definida.

1.6.g. Modifica la interrupcion 0x46

Se implementa el handler de la interrupción 0x46, que se va a encargar de atender interrupciones de los piratas, con el objetivo de realizar las distintas acciones de las que puden hacer uso los piratas: moverse, cavar y dar su posición. En cada caso, se llama a una función en C que se encarga de resolver estas tareas. Comparamos el valor del registro eax con 0x1, 0x2 o 0x3 y saltamos a la función correspondiente. Si además recibimos un parámetro en ecx, lo pusheamos para que pueda ser usado por la misma. Al final de la interrupción, se salta a la idle. Siempre es posible realizar el salto ya que esta interrupción solo puede ser generada por un pirata.

- game_syscall_pirata_mover: si la posición a la que el pirata se quiere mover es válida, se realiza el movimiento del pirata. Es decir, si es minero, se mapea la posición a mover con la 0x400000, y se copia el código. Luego se actualiza la posición actual del struct del pirata. Si es explorador, además de esto, se mapean las posiciones de alrededor, para él y todo el resto de los piratas, así los mineros pueden desplazarse para buscar los tesoros. Desde esta función se llama a las que se encargan de pintar los elementos en la pantalla.

- game_syscall_cavar: esta función, como es llamada en el caso de cavar, solo se corre cuando un minero es el pirata actual. Si este esta parado sobre una posición con un botín disponible (con un valor mayor a cero), entonces se disminuye en uno el valor del botín y aumenta el valor valor de los puntos del jugador. También reinicia el reloj global, para que no termine la partida todavía. Caso contrario, de no haber botín, el minero muere.

- game_syscall_pirata_posicion: devuelve la posición del pirata actual con la codificación $y << 8|x$. Ponemos este valor en la posición correspondiente de la pila para que el mismo quede en el registro eax, respetando convención C.

1.6.h. Salto a una tarea pirata

Para este ejercicio temporal se hizo el salto manualmente a una tarea, en particular al primer explorador del jugador A. Previamente se rellena, para esta tarea, la entrada de la gdt y su tss, ya que el scheduler todavía no está implementado, y se inicializa la instancia del struct. En el kernel se carga en la variable *selector*, previamente definida, el valor 0x78 y se hace $jmp\ far\ [offset]$.

1.7. Ejercicio 7

1.7.a. Estructuras del Scheduler

Para el intercambio de tareas mediante el scheduler utilizaremos dos arreglos 'tareas_A' y 'tareas_B' de booleanos en donde la i-ésima posición será true si esa tarea pirata se encuentra "viva" en juego.

La función *inicializar_estructuras_sched* seteará en false todas las entradas de ambos arreglos, dado que al inicio no se deberá ejecutar ninguna tarea pirata.

Además contamos con una variable global *jug_actual* que indica el jugador cuya tarea está siendo ejecutada actualmente. Por defecto al iniciar el scheduler se setea al jugador A.

Por último, se cuenta con otras dos variables globales que indican el índice en los arreglos de booleanos mencionados anteriormente de la tarea que está siendo ejecutada por el jugador actual y el de la última ejecutada por el otro jugador. Por defecto se inicializan en 0.

1.7.b. Próxima tarea del Scheduler

Implementamos la función *prox_tarea_a_ejecutar* que devuelve el índice en la gdt de la próxima tarea en ser ejecutada.

Primeramente la función se fija en ambos arreglos de booleanos si se da el caso en que no hay ninguna tarea activa o si el juego ya terminó, en ambas situaciones devuelve el índice de la gdt de la tarea idle.

Caso contrario, comprueba si hay al menos una tarea activa del jugador que no es el actual, en ese caso, actualiza la variable *jug_actual* y devuelve el índice de la gdt de la próxima tarea de dicho jugador. Si no, devuelve el índice de la próxima tarea del jugador que se encontraba ejecutando una hasta ese momento, dejando con el mismo valor la variable *jug_actual*.

Para obtener el índice de la próxima tarea a ser ejecutada, usamos el valor de la variable que indica la última tarea en ser ejecutada: recorremos el arreglo de booleanos desde la siguiente posición hasta el final buscando una entrada que sea true, si no se encuentra, se recorre desde el inicio del arreglo hasta la posición marcada por la variable global. Sabemos que habrá por lo menos una entrada en todo el arreglo con valor true.

1.7.c. Sched_tick

La función *sched_tick* obtiene el índice de la gdt correspondiente a la próxima tarea a ser ejecutada, se fija si es el de la idle, y en caso de no serlo, llama a la función *game_tick* con el índice correspondiente, y el jugador actual.

Dentro de *game_tick* se actualizan los valores correspondientes a las variables globales *id_pirata* (que contiene el índice de la gdt de la tarea que está por ejecutarse) y *jugador_actual*. Si hay algún minero que deba ser lanzado, se llama a la función *game_jugador_lanzar_pirata* con el jugador correspondiente y el valor *true* que indica que tiene que inicializarse un minero. Luego chequeamos si fueron recolectadas todas las monedas del juego, o la variable contador (que se va restando cuando están las 16 tareas activas a la vez) llegó a 0, y terminamos el juego en cualquiera de los dos casos.

La función *game_jugador_lanzar_pirata* setea todos los campos correspondientes en un struct pirata dentro del arreglo de piratas del jugador pasado como parámetro. Allí se completan los siguientes campos: activo en *true*, jug con el puntero al jugador que tendrá ese pirata, indice_arreglo_jug con la posición del pirata en el arreglo de piratas del jugador, indice_gdt con el índice correspondiente de la tarea en la gdt, posición actual con la posición del puerto que corresponde, tipo con *true* si es un minero y $false$ si es explorador. Luego inicializa la tss y el segmento de la gdt correspondiente a la tarea, pinta al pirata en la posición del puerto y luego, si es un minero, le pasa las componentes x e y del botín que debe buscar. Para eso tomamos el cr3 de la tarea que acabamos de inicializar, lo cargamos con la instrucción *lcr*3 y copiamos en las posiciones de memoria que se encuentran debajo de la pila dicha posición: el valor de x en $0x400000 + 0x1000 - 8$ y el de y en $0x400000 + 0x1000 - 4$, también en la componente tesoro del pirata, y volvemos a cargar el cr3 que se encontraba antes de haber copiado esas posiciones.

1.7.d. Int 0x46

Este ejercicio es similar al 1.6.g

1.7.e. Intercambio de tareas en cada ciclo del reloj

Para esto tomamos el valor actual del task register con la instrucción *str*, comparamos su valor con el devuelto por *sched_tick*, si es igual, no hacemos nada y seguimos ejecutando la misma tarea. Si no, movemos el valor devuelto a la variable *sched_tarea_selector* y hacemos un jmp far para cambiar de tarea.

1.7.f. Modificar rutinas de excepciones

En el caso en que ocurra una excpeción, llamamos a la función *matar_pirata* que se encarga de matar a la tarea actual (que acaba de generar la excepción) poniendo en false la posición correspondiente en el arreglo de tareas del scheduler. Luego de eso saltamos a la tarea idle.

1.7.g. Modo Debug

Para el modo debug tenemos una variable global llamada *modo_debug* que inicialmente tiene el valor 0. En caso de apretar la tecla 'y', se cambia su valor a 1 y si luego ocurre alguna excepción se mostrará en pantalla el modo debug.

Para eso en las excepciones del procesador comprobamos si el valor de esta variable se encuentra en uno y, de ser así, guardamos el estado de la pantalla actual, imprimimos todos los registros correspondientes (que fueron pusheados a la pila y por lo tanto pasados como parámetros a la función que usamos para mostrarlos al iniciarse la interrupción y al hacer el pushad) y nos quedamos ciclando hasta que se vuelva a apretar la tecla 'y', en cuyo caso restauramos la pantalla a su estado anterior al modo debug, matamos la tarea que generó la excepción y saltamos a la idle.