DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Informe - Grupo: SnakeII/Nokia1100

21 de julio de 2015

Organización del Computador II
Trabajo Práctico Nro. 3

Integrante	LU	Correo electrónico
Pablo Gomez	156/13	mago-1986@hotmail.com
Lucía Parral	162/13	luciaparral@gmail.com
Petr Romachov	412/13	promachov@gmail.com

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

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) 4576-3300
http://www.exactas.uba.ar

Índice

1. Introducción

1.1. Introducción

El objetivo de este trabajo práctico es aplicar conceptos de System Programming al crear un sistema mínimo que permita correr hasta 16 tareas de manera concurrente a nivel de usuario, capaz de capturar cualquier problema que estas tareas puedan generar y tomar las acciones necesarias para desalojarlas. En este marco, implementamos un juego de dos jugadores, que tienen la posibilidad de lanzar exploradores hacia un mapa. Estos exploradores son tareas que se mueven por la memoira en busca de botines para consumir y acumular puntos, por lo que se necesitan piratas mineros que desentierren dichos botines. Cada jugador puede lanzar hasta 8 piratas entre exploradores y mineros. Los mineros liberan su slot tras desenterrar botines mientras que los exploradores quedan "vivos para siempre". Los piratas son capaces de moverse, de consultar su posición y la de los otros de su equipo y de excavar para desenterrar tesoros. Cada vez que desentierran un tesoro, ese botin pierde una moneda hasta quedar vacío. El juego termina cuando se agotan todos los botines o bien todos los slots de piratas estén ocupados y los piratas actuales no sean capaces de desenterrar los botines, donde en dicho caso, el juego acaba tras un tiempo prudencial. En ambos casos, gana el jugador con más monedas.

2. Ejericio 1

2.1. Completar la tabla de descriptores globales

Para esto completamos el indice 0 de la GDT con el descriptor nulo. Las siguientes 7 entradas por requerimientos de este trabajo debian ser nulas. A continuación, completamos con entradas para código y datos para sistema operativo (nivel 0) y para usuario (nivel 3). Los segmentos de código y datos del sistema operativo están descritos en las entradas 8 y 9 respectivamente, mientras que los de usuario en las entradas 10 y 11. El campo límite para estas entradas es 0xF3FF y la base 0x0000, con granularidad 1, ya que direccionan 500MB de memoria. El tipo para los los descriptores de código es 0x8, correspondiente a código de sólo-ejecución, mientras que para los de datos es de 0x2, para segmentos de datos de lectura/escritura. También es relevante indicar que para todas estas entradas, el campo p es 1, ya que los segmentos se encuentran presentes. También tenemos una entrada de datos para el segmento de video, con base 0xB800 y límite 0xC0000, con dpl 0, pues es para ser utilizado sólo por el kernel, presente y permisos de lectura/escritura. Este descriptor tendrá índice 12.

2.2. Pasaje a modo protegido

En primer lugar, cambiamos el modo de video a 80x44. Habilitamos a20 llamando a `habilitar_A20`. Cargamos la GDT con la instrucción `lgdt`.

Para pasar a modo protegido, seteamos en 1 el bit PE del registro CR0 y hacemos jump far con un selector de segmento de código de nivel 0 y offset `modo_protegido` (etiqueta que declaramos a continuación)

Una vez en modo protegido, asignamos los selectores de segmentos. Como ds, ss elegimos segmento de datos de nivel 0, y para fs el segmento de video.

Seteamos la pila del kernel en la direccion 0x27000.

2.3. Inicialización de la pantalla y juego

Usamos las funciones game_`inicializar`, `screen_inicializar` y `game_inicializar_botines`.

La función `screen_inicializar`, que pinta el fondo de gris, y el recuadro para cada jugador en azul y rojo. `game_inicializar` arranca todas las variables del juego para su uso llamando a `game_juagdor_inicializar` para cada jugador. A su vez, esta última función completa la estructura del jugador correspondiente. `game_inicializar_botines` dibuja todos los botines en la pantalla

3. Ejericio 2

3.1. Creación de las estructuras de la IDT

En la función `idt_inicializar` inicializamos las entradas iniciales de la IDT (de la 20 a la 31 estan reservadas para Intel). Ademas inicializamos las entradas 32 para reloj, 33 para teclado y 70 para las syscalls a ser utilizadas por las tareas. Las entradas de la IDT estan definidas por una macro en la que elegimos el número que representa su índice. A la entrada para la syscall le damos los siguientes atributos: 0xEE00. A todas las demás: 0x8E00.

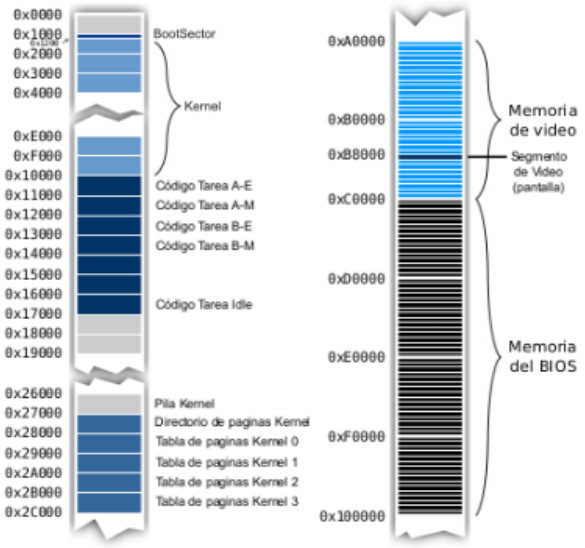
Estas difieren ya que la syscall al ser llamada por la tarea, necesita tener dpl = 3 para que esta pueda accederla, mientras que las otras interrupciones no.

Luego para que el procesador utilice la IDT utilizamos la interrupción lidt y configuramos el controlador de interrupciones con las funciones `resetear_pic` y `habilitar_pic`, provistas por la cátedra.

4. Ejericio 3

4.1. Inicializar directorio y tabla de páginas

Se pide generar un directorio y tablas de páginas para el kernel (`mmu_inicializar_dir_kernel`). Para esto se debe generar un directorio de páginas que mapee, usando identity mapping las direcciones 0x00000000 a 0x003FFFFF. El directorio debe estar inicializado en la dirección 0x27000 y las tablas de páginas según muestra la figura.



Para esto, en la función `mmu_inicializar_dir_kernel`. Seteamos la direccion de la page directory en 0x27000 y de la page table 0 en 0x28000 como indica la figura. Luego llamamos a `mmu_empty_mapping` que se encarga de inicializar las entradas 1 a 1024 de la page directory con base 0x00000 y atributos 0x00. y a la función `mmu_identity_mapping` que se mapea con identity mapping las entradas 0 a 1024 de la page table, con atributos p=1 y r/w=1. La entrada 0 de la page directory tiene como base la dirección de la page table 0 (0x28000) y atributos p=1 y r/w=1.

Una vez creadas estas estructuras, habilitamos la paginación cargando en cr3 la dirección del page directory (0x27000) y seteando en 1 el bit más significativo de cr0.

5. Ejericio 4

5.1. Manejo de memoria para las tareas

En este ejercicio, con el fin de poder manejar la memoria utilizada por las distintas tareas, creamos tomando como base las sugerencias dadas en el enunciado, las siguientes funciones que permiten inicializar un directorio de páginas para una tarea, así como mapear y desmapear páginas.

Rutina `inicializar_mmu` para administrar la memoria en el área libre. Esta función setea la variable global next_page en la primer página no usada (`AREA_LIBRE` = 0x100000).

`mmu_inicializar_dir_pirata` donde pedimos una nueva pagina para el directorio de páginas y seteamos en su primer entrada la tabla del kernel con p=1 y r/w=1. Luego mapeamos la pagina de codigo (0x400000) con la posicion fisica del pirata en el mapa y copiamos el codigo pertinente del pirata que corresponde. Además se mapean todas las posiciones ya exploradas por el jugador.

Para esto inicializamos una page directory nueva, ya que cada tarea dispondrá de la suya propia. Con este fin, en primer lugar pedimos una página libre con `mmu_new_page` que da la dirección a la próxima página a mapear libre. Luego hacemos empty mapping sobre sus entradas. La primer entrada será usada para mapear el kernel, por lo tanto pone el código como su base, la setea en presente y con permisos de supervisor. A continuación mapeamos la página del código. Para esto vamos a cambiar el cr3 (previamente lo guardamos para no perderlo haciendo uint cr3 = rcr3) y hacemos `mmu_mapear_pagina` pasando como parámetros la dirección en la que va a estar el código del pirata (0x400000) y el área del mapa en la cuál está la tarea. Es decir, mapea el área del mapa en donde está la tarea a 0x400000. De esta forma, $game_xy2lineal(p->coord.x, p->coord.y) * PAGE_SIZE + MAPA_BASE_FISICA$ es nada más una traducción a la dirección física del mapa correspondiente a la posición (x, y) del pirata.

`mmu_copiar`recibe una dirección fuente y una destino (como punteros a unsigned int) y copia el contenido de una en la otra. Esto se concreta a través de un simple "for"que itera por ambas páginas, asignando fuente[i] a destino[i]. Como el tamaño de una página es 4096 y se va copiando de a 4 bytes, el número de iteraciones es 1024. Con esta función que recibe como dirección fuente el código específico de la tarea y como dirección destino la 0x400000 copiamos el código a la dirección mapeada previamente.

Por ultimo se mapean todas las pocisiones del mapa que fueron exploradas por el jugador específico al pirata.

`mmu_mapear_pagina` permite mapear la página física en la dirección virtual pasada como parámetros, utilizando cr3. Para esto, recibimos el cr3 y a partir de este obtenemos la dirección de la page directory. Del parámetro virtual obtenemos el offset para la entrada en la page directory y de la page table. Si la entrada de page directory no está presente, creamos una nueva página y para esta entrada, hacemos `empty_mapping` sobre las 1024 entradas, es decir, las seteamos como no presentes. Luego, para la entrada correspondiente al offset de page directory, ponemos como base la dirección de la página creada, y atributos 0x03. Finalmente, obtenemos la dirección de la tabla creada y para el offset calculado inicialmente, ponemos como base la dirección física, y como atributos permisos de usuario, presente habilitado y lectura/escritura o solo lectura según corresponda.

`mmu_unmapear_pagina` borra el mapeo creado en la dirección virtual utilizando cr3. Par esto, como en `mmu_mapear_pagina`, calculamos el offset de directorio y tabla a partir de la dirección virtual. Luego, si la entrada correspondiente no existe, significa que ya se encontraba desmapeada y no hacemos nada. Caso contrario, limpiamos las entradas de page directory y page table correspondiente, seteando en 0 el bit de presente.

6. Ejericio 5

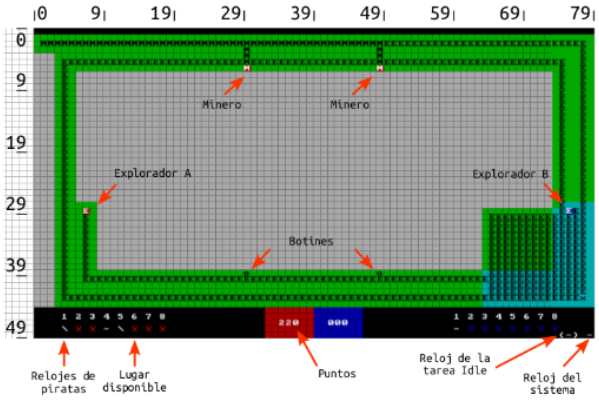
6.1. Rutinas de atención de interrupciones

En este punto, se nos encarga desarrollar las rutinas de atención de interrupciones para las distintas entradas de la IDT, en particular, la del reloj, la del telcado y la 0x46.

Como en este punto se nos pide el desarrollo de algunas funcionalidades que en los puntos posteriores son reemplazadas por otra, desarrollaremos en este punto el resultado final de las distintas rutinas, tal como se ven en el código terminado.

6.1.1. Rutina de interrupción del reloj

Cada vez que se llama a la interrupción del reloj ante cada tick, se llama a la función `sched_tick`, quien realiza un llamado a game_tick y además devuelve lo obtenido tras el llamado a `sched_proxima_a_ejecutar` que es la próxima tarea a ejecutarse. La función `game_tick` llama a `screen_actualizar` que pinta en pantalla los puntos de cada jugador, además de mostrar el reloj global del juego rotando, y el reloj particular del pirata actual, rotando debajo de su número indicador correspondiente, como indica la figura. Para los piratas no activos o muertos, se muestra una cruz.



Además, si el jugador tiene mineros disponibles (es decir, tiene mineros para mandar, que se acumularon después de que un explorador descubriese un botín) y la cantidad de piratas vivos del jugador actual es menor a 8, se lanza un pirata minero con la función `game_jugador_lanzar_pirata`. En esta función se inicializa un pirata a través de `game_pirata_inicializar` que primero busca cuál es el próximo id disponible para el pirata de un jugador, y luego inicializa los campos del struct pirata. Después se inicializan las entradas de TSS y GDT correspondientes para el pirata a lanzarse con las funciones `tss_inicializar_pirata` y `gdt_inicializar_pirata` descritas en detalle en el punto siguiente y se pinta al pirata en pantalla con la función `screen_pintar_pirata`. Luego actualizamos las variables del juego, aumentando la cantidad de piratas del jugador, y si el pirata lanzado es un explorador se aumenta la cantidad de exploradores del jugador actual Si es minero cambiamos el cr3 actual por el cr3 del minero obtenido de su TSS para obtener del stack del minero las direcciones destinadas a botín x e y, y las actualizamos con la posición del botín del jugador. Finalmente, se restaura el viejo cr3 y decrementa en uno la cantidad de mineros disponibles del jugador, ya que ahora tiene uno menos para mandar.

En la función `sched_proxima_a_ejecutar` utilizamos la función auxiliar `sched_proximo_pirata`, que recibe un id de jugador y id de pirata y se fija para ese jugador, cual es el siguiente al pirata dado que está vivo y lo devuelve. En caso de que no haya ninguno, se devuelve -1.

Luego, se chequea, en primer caso, si no hay ninguna tarea activa para ningún jugador (es decir, el llamado para `game_proximo_pirata` para ambos jugadores con el pirata actual dio -1), devuelve el índice en GDT de la tarea Idle.

Sino, se llama a `game_proximo_pirata` con ambos jugadores y el pirata actual y se determina para cada jugador, cual es el próximo pirata a ejecutarse, en caso de que tenga uno, y se setea la proxima tarea a ejecutarse según cual sea el jugador actual (en este caso, la siguiente del otro jugador) o según el otro jugador no tenga tareas para ejecutar (en este caso, la siguiente del jugador con tareas).

Finalmente, con lo devuelto por `sched_tick` (el indice en GDT de la próxima tarea a ejecutarse) seteamos el valor del selector en [sched_tarea_selector] y realizamos un jmp far a [sched_tarea_offset] definidos previamente como sched_tarea_offset: dd 0x00 sched_tarea_selector: dw 0x00 en la sección de datos, completando el task switch.

6.1.2. Rutina de atención del teclado

Para la atención del teclado, en la rutina leemos la tecla que fue presionada y con esta llamamos a `game_atender_teclado`, que en caso de recibir la tecla Shift Left o Shit Right lanza al pirata correspondiente. Además, en caso de que sea la tecla Y, setea el modo debug según corresponda.

6.1.3. Rutina de atención de las syscalls (_isr70)

En la rutina `_isr70` llamamos a la función `game_syscall_manejar` con el número de syscall recibida, para luego realizar las distintas rutinas según sea necesario. Las distintas rutinas son:

1. `game_syscall_pirata_mover` que recibe como parámetros un id de pirata y una dirección a la cual debe moverse, que se basa en un enum que tiene los valores ABA, ARR, IZQ, DER. Para poder manejar la dirección pasada como parámetros de la misma forma en la que tratamos a las coordenadas de los piratas, convertimos la dirección a un struct coord con la función `game_dir2coord`. Una vez realizada la conversión, con la función `game_posicion_valida`, comprobamos que la posición a la que queremos movernos esté dentro de los límites del mapa. Si no lo está, retornamos sin realizar ningún movimiento. Caso contrario, actualizamos la posición del pirata. Si el pirata es explorador, y la posición no había sido previamente explorada por éste, llamamos a `game_habilitar_posicion` que es una función que para cada posición alrededor del explorador, incluyendo a la actual, si ésta no fue explorada la marca como explorada y la mapea. Además, chequea si hay tesoro en esa posición, y de ser así, incrementa en uno la cantidad de mineros del jugador ya que en el próximo `game_tick` deberá lanzarse un minero. Además se guarda en el botín del jugador la coordenada recién habilitada.
2. `game_syscall_cavar` recibe un id de pirata. Si este no corresponde a un minero, retornamos sin realizar ninguna acción. Si es minero, llamamos a `game_buscar_botin` con el pirata para ver si en la posición dada hay un botin. Si lo hay pero el valor del tesoro, calculado con `game_valor_tesoro` es 0, se libera al minero, mientras que si el valor es mayor a 0, se suma uno a la cantidad de monedas del jugador que mando la syscall y se resta uno al botín de la posición.
3. `game_syscall_pirata_posicion` recibe como parámetro el id del pirata que llama a la syscall y el id del pirata del cuál se quiere conocer la posición. Si el id del pirata del cual se quiere conocer la posición es -1, se devuelve la posición del jugador que llama a la syscall, caso contrario, se devuelve la posición del pirata cuyo id fue pasado por parámetro.

7. Ejericio 6

7.1. Inicialización de descriptores de TSS en GDT

En este punto, se nos pide definir las entradas en la GDT para ser usadas como descriptores de TSS a ser utilizadas para las tareas. En la entrada 14 tenemos el descriptor de TSS de la tarea inicial. Para este, seteamos como límite 0x67 y como base, la dirección `&tss_inicial`, con tipo 0x09 (Execute only accessed), el bit de presente habilitado, dpl 0 y granularidad 0. En la entrada 15, el descriptor de la TSS de la tarea idle, tiene los mismos atributos, excepto que la base es `&tss_idle`. Para la inicialización de las entradas de la GDT correspondientes a las tareas piratas, creamos una función gdt_inicializar_pirata que dado un un indice y una dirección de TSS, crea una entrada de gdt en la posición gdt[índice] que tiene como base la dirección de TSS, dpl 3, tipo 0x09 (Execute only accessed), límite 0x67, bit de presente en 1 y granularidad 0. Esta función será llamada al momento de lanzar un pirata con el offset correspondiente al jugador y número de tarea a lanzar, y la dirección de la tss inicializada previamente en la función `tss_inicializar_pirata` que será descrita a continuación.

7.2. Inicialización de TSS

A continuación, se nos pidió completar la entrada de TSS correspondiente a la tarea Idle. Para esto implementamos la función `tss_inicializar` que completa los distintos campos, entre ellos el cr3, cargando el cr3 actual, los flags activados, el eip con la dirección 0x16000, que es donde según el enunciado se encuentra la tarea idle. El esp y ebp en 0x27000, dirección donde se encuentra el directorio de páginas de kernel, y donde cs es el segmento de código de nivel de supervisor, es, dd, ds, gs el segmento de datos de nivel supervisor y fs es el segmento de video.

Luego, para completar una TSS libre con los datos de una tarea utilizamos la función `tss_inicializar_pirata` mencionada anteriormente, en la quedado un id de jugador y un id de pirata, obtenemos un puntero a la tss correspondiente a dicha tarea y completamos sus campos. Los flags activados, como eip la dirección 0x400000 (CODIGO_BASE), que es donde se encuentra el código respectivo de la tarea, esp es `CODIGO_BASE+PAGE_SIZE-12` y ebp `CODIGO_BASE+PAGE_SIZE-12`. El campo cs corresponde al selector de segmento de código de nivel de usuario, mientras que es, ss, ds y fs al de datos de nivel de usuario. Todos los registros son inicializados en 0. Para el cr3 se llama a `mmu_inicializar_dir_pirata`, que fue detallada anteriormente y para esp0 a se pide una página para la pila de nivel 0 con `mmu_new_page` . Ss0 corresponde también al segmento de datos a nivel supervisor. Como anticipamos, esta función es llamada desde `game_jugador_lanzar_pirata` antes de inicializar el descriptor de GDT de dicha tarea.

7.3. Task switch entre tarea inicial e Idle

Para saltar intercambiando tareas entre la inicial y la idle, en kernel.asm primero definimos en la sección de datos a selector y offset como un word y double word respectivamente. Antes de pasar a la tarea idle, es necesario cargar la tarea inicial, haciendo uso de la instrucción `ltr` que la carga en el task register. Luego para hacer el intercambio de tareas entre la inicial y la idle, escribimos un código que mueve el selector de segmento correspondiente a la entrada de la gdt de la tarea idle a [selector] (índice 14) y luego hacemos un jmp far a [offset]. El jump far produce el task switch que permite realizar el intercambio de tareas.

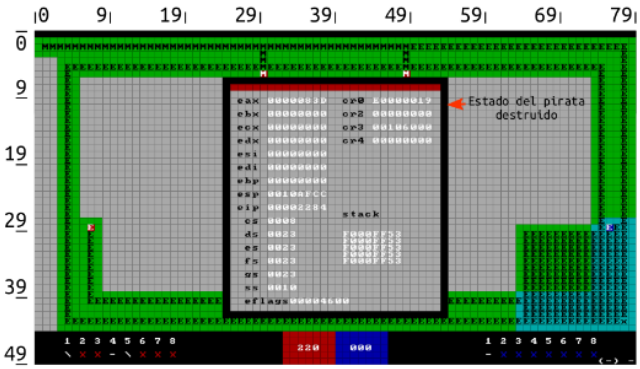
8. Ejericio 7

8.1. Scheduler, debugger y etcs

En este punto, se nos pide inicializar las estructuras necesarias para el scheduler. Para esto, tenemos la función `sched_inicializar` que setea como jugador actual al jugador A, y como pirata actual de ambos jugadores al 0.

Las funciones `sched_tick`, `game_proximo_pirata` y `sched_proxima_a_ejecutar`, también implementadas en este punto, fueron detalladas anteriormente, así como también el desarrollo de la rutina de la interrupción 0x46 para que atienda los servicios de mover, cavar y dar la posición de un pirata.

Por último, implementamos el mecanismo de debugging pedido en el enunciado, en el que al presionar la tecla Y se accede al modo debug, y en cuanto se produzca una excepción se mostrará en pantalla el estado del procesador tal como se muestra en la figura.



Al presionar nuevamente la tecla Y, se borra la información presentada en pantalla y se desaloja la tarea que produjo la excepción y se salta a la tarea Idle hasta que se decida en el próximo ciclo de reloj cual es la próxima tarea a ser ejecutada. Para realizar esto, en la macro realizada para las excepciones de la 1 a la 19, se chequea si nos encontramos en el modo debug. De no ser así, retornamos sin hacer nada. Si estábamos en modo debug, se salva el estado actual de la pantalla con `screen_pantalla_debug` y se queda ciclando hasta que se detecte que se presionó nuevamente la tecla Y. Cuando esto suceda, se llama a `load_screen` que restablece la pantalla previa y a continuación se llama a `game_pirata_exploto`, que setea al pirata actual como muerto, y resta uno a la cantidad de piratas del jugador correspondiente. A su vez, setea el bit de presente del descriptor en GDT del pirata muerto como 0 y actualiza el reloj del pirata en la pantalla con `screen_actualizar_reloj_pirata`. Finalmente se setea como próximo selector al de la tarea Idle y se realiza un jmp far a esta. El flag de modo debug se setea en 0 o 1 (de acuerdo al valor previo del flag) en `game_atender_teclado` cuando se detecta que la tecla presionada fue Y. La función `screen_pantalla_debug` salva el estado inicial de la pantalla con `save_screen` y printea en pantalla la pantalla de debug según se muestra en la ilustración, con los datos correspondientes para la tarea que lanzó la excepción. Como cuando se lanza una excepción de una tarea nivel que está corriendo en nivel 3 se cambia el nivel de privilegio y este pasa a ser 0, se guardan en el stack varios parámetros que usaremos para mostrar en pantalla de debug, ya que estos argumentos obtenidos de la pila son usados como sus parámetros. Además, como utilizamos la instrucción pushad antes de llamar a la función, guardamos en la pila todos los registros de propósito general.