DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico III

## El Kernel contraataca

Organización del Computador II

Segundo Cuatrimestre de 2014


| Integrante | LU | Correo electrónico |
|---|---|---|
| Alejandro Mignanelli | 609/11 | minga_titere@hotmail.com |
| Franco Negri | 893/13 | franconegri2004@hotmail.com |
| Federico Suárez | 610/11 | elgeniofederico@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

**Resumen**

En el presente trabajo se describe el desarrolo del Kernel desarrollado para una arquitectura intel de 32-bits, así como el manejo de paginación, manejo de tareas, interrupciones y todo lo referente al manejo de un pequeño sistema operativo.

# Índice

# 1. Objetivos generales

El objetivo de este trabajo practico, partiendo de un procesador intel de 32-bits, generar un kernel capaz de gestionar memoria entre diferentes tareas, correrlas de manera concurrente, y resolver las diferentes problematicas que puedan surgir al momento de ejecución.

Para ello utilizaremos las diversas herramientas que intel pone a nuestra disposicion en modo protegido: Usaremos segmentacion y paginación para controlar el privilegio con el que las tareas se ejecutarán, ademas de limitar lo que las tareas puedan 'ver' con un mapeo parcial de la memoria. Utilizaremos interrupciones del procesador que permitirán, tanto reaccionar de manera apropiada cuando se produzca un error en tiempo de ejecución, obtener input del teclado y gestionar un task manager que nos perimita ejecutar tareas de manera concurrente.

En el presente informe, se detallará de manera mas elaborada todo lo hecho para conseguir el objetivo del trabajo práctico, asi como cualquier decisión que se haya tomado en el código a tales fines. Para su mejor entendimiento, este informe se dividirá en ejercicios, que son pequeñas partes del trabajo, y todos juntos conforman al trabajo práctico en si.

# 2. Ejercicio 1:

En este primer ejercicio, decidimos completar las primeras 7 posiciones de la gdt en un estado claramente inválido para que, en caso de usarlos erroneamente en algun momento, el error fuera visible. Luego, pusimos 4 segmentos, dos para código de nivel 0 y 3, y dos para datos de nivel 0 y 3. Para mayor entendimiento del código, se han usado defines, con nombres declarativos(por ejemplo, el segmento destinado a codigo de nivel 0, se llama GDT_IDX_CDE_LVL_0). Estos segmentos direccionan los primeros 623 MB de memoria. Tambien se ha declarado un segmento que describe el area de la pantalla en memoria que puede ser utilizado solo por el kernel. Esto se utilizará en principio para escribir en el buffer de memoria, pero más avanzado el trabajo, será necesario mantener muchas estructuras de datos que deberán ser impresas por pantalla, por lo tanto se abandonará este método y se pasará a un codigo en C, mas limpio y fácil de mantener. Dado que la convención C nos pide que todos los segmentos de datos apunten al mismo segmento, este segmento terminará quedando en desuso y se utilizará para escirbir en pantalla tan solo el segmento de datos de nivel 0. Después, para pasar a modo protegido, deshabilitamos las interrupciones, habilitamos el A20, cargamos la GDT, seteamos el bit PE del registro CR0 en 1 para habilitar modo protegido, y luego hacemos un jump far a un segmento válido para pasar efectivamente a modo protegido. Una vez hecho el pasaje, seteamos todos los segmentos para que apunten al mismo descriptor de segmento (pedido por la convención c) y seteamos la base y puntero de la pila del kernel en la posición 0x27000. Finalmente inicializamos en pantalla la interfaz gráfica que tendrá el juego, para lo cual utilizamos el segmento anteriormente mencionado.

# 3. Ejercicio 2:

En este ejercicio inicializamos la IDT, es decir, completamos las entradas necesarias para asociar diferentes rutinas a todas las excepciones del procesador. En principio, cada una de estas rutinas mostrará por pantalla qué tipo de problema se produjo e interrumpirá la ejecución. Luego de inicializada la IDT, deberemos cargarla desde el kernel. Posteriormente se modicarán estas rutinas para que en caso de que sea una tarea de nivel 3 se resuelva el problema y se desaloje a la tarea que lo produjo. En caso de producirse un error de kernel, se mostrará a que tipo pertenece y en caso de tener, su code error.

# 4. Ejercicio 3:

Inicializamos el directorio de páginas del kernel y las tablas de páginas necesarias para mapear las direcciones 0x00000000 a 0x003FFFFF usando identity mapping para ello usamos un pequeño código en asambler que puede verse en $paging.asm$. En principio esta función era suficiente para el simple mapeo que requeríamos. Mas adelante se optará por inicializar los page directories y page tables de una manera

más clara y limpia utilizando $mmu.c$. El directorio de páginas se inicializará en la dirección $0x27000$ y las tablas de páginas a partir de la dirección $0x28000$. En este paso, se implementa una función llamada pedirPag, que dada la dirección de una página, la inicializa con todas sus entradas en cero, poniendo exclusivo cuidado en setear en todas sus entradas el bit ”not present”de manera que no sea utilizada hasta que se determine una dirección apropiada para la misma. Luego de armado el directorio de pagias del kernel, seteamos el CR3 para que apunte al page directory y seteamos en 1 el bit correspondiente del registro CR0 para habilitar la paginación.

# 5. Ejercicio 4:

Aquí implementamos la Memory Managment Unit dentro de $mmu.c$, para ello, primero la inicializamos seteando una variable ”dirGlobal”la cual siempre apuntará a una dirección física libre de memoria que será utilizada para alojar una page table o page directory según corresponda.

Luego construimos una función para inicializar zombies ($mmu\_inicializar\_zombie$) que recibe como parámetros el código de la tarea y un arreglo de direcciones físicas(éste contiene como primer elemento a la posición en el mapa donde estamos inicializando el zombie, y en el resto del arreglo las 8 posiciones del mapa contiguas). En esta función creamos un nuevo directorio de páginas y una tabla de páginas para la tarea zombie y vinculamos la tabla de páginas del kernel a la primera entrada del directorio de páginas de la tarea zombie. Después, mapeamos cada dirección del arreglo a partir de la dirección virtual 0x8000000. Dado que el area del mapa donde se ejecutarán los zombies no es alcanzable desde el kernel deberemos mapear la dirección física donde se encontrará el zombie en alguna dirección virtual que no utilicemos de manera tal que podamos copiar su código a esta dirección, y una vez copiado el código, desmapearlo.

Para eso, agregamos al page directory actual una nueva page table, mapeamos en ella una dirección que no se use (en este caso se decidió $0xDC4000$), copiamos el código, y desligamos la page table del page directory.

Para todo este proceso se implementarán varias funciones(creo que habría que poner algunas de las funciones y explicar minimamente que hacen(ALE)) que facilitarán la tarea de mapear y desmapear page tables a direcciones reales y page directories a page tables, que más adelante cuando sea necesario remapear zombies, resultarán de suma utilidad.

# 6. Ejercicio 5:

Completamos las entradas necesarias en la IDT para asociar las rutinas correspondientes a las interrupciones del teclado, reloj y la interrupción de sofware 0x66. De momento estas funciones no tendrán mayor utilidad, el teclado solo mostrará un mensaje por pantalla al presionarse una tecla y el reloj solo actualizará un reloj en la parte inferior derecha de la pantalla, pero estos comportamientos nos permiten poder probar que las interrupciones se realizan correctamente.

De momento la interrupción 0x66 no realizará ninguna acción, se hablará de como fue implementada mas adelante en el tp.

Posteriormente se reescribirán dichas rutinas para que, la del reloj sea utilizada para actualizar la pantalla y llamar al skeduler y cambiar a la tarea adecuada en cada tick del reloj, la del teclado que será utilizada para que las distintas teclas respondan a la función asignada y finalmente la interrupción 0x66 que se utilizará para el movimiento de las tareas zombies.

# 7. Ejercicio 6:

En este ejercicio se implementará principalmente el archivo $tss.c$ el cual se encargará de administrar de buena manera todo lo que tenga que ver con el seteo apropiado de las tss a ser utilizadas.

En principio se crea una entrada vacía que será donde guardemos la tss del kernel en el momento en que este deje de ser de utilidad.

Como segundo paso se setea de manera apropiada las variables de la tarea idle en $inicializarTareaIdle()$.

Luego se implementará una función $cargarTSS\_zombie()$ que se encarge de inicializar 16 entradas en la gdt que corresponderán a cada uno de los zombies que pueden estar activos en el tablero. Esta función primero setea las entradas de la gdt para que apunten unívocamente a un tss a ser utilizada por un zombie y luego se setea la tss correspondiente tambien en estado válido para que queden listas para ser utilizadas por una tarea de nivel 3, lo único que quedará sin setear será el CR3 de las tss, eso, se determina,(ALE) será agregado en el momento en que $mmu\_inicializar\_zombie$ nos dé un CR3 valido.

Además, para simplificar las cosas, se tomará como que las primeras 8 tss son de los zombies del jugador A y las siguientes 8 entradas son del jugador B. De esa manera con tan solo tener el indice de que tss se está corriendo, podremos además determinar a que jugador pertenece.

Más adelante veremos como setear las tss de los zombies en este momento, será inecesario para la implementación final, por lo que esta funcionalidad será quitada de esta función y se moverá a otro lado(ATENTOS CON MI IGNORANCIA, este parrafo dice lo que quiere decir?? solo me suena raro, por eso pregunto, cualquier cosa borren este comentario(ALE)).

# 8. Ejercicio 7:

En este ejercicio se implementa un scheduler simple que a partir de lo devuelto por la función $sched\_proximo\_indice()$ (implementada en $sched.c$). Por ahora y dado que no hay otras tareas para correr, este scheduler siempre retornará el valor de la tarea idle, pero más tarde se agregará la lógica para que además alterne entre zombies en el caso en que sea necesario.

En este momento del tp, todavía no contamos con la lógica para saber donde se encuentra un zombie, como será la lógica para mapearlo, quien controlará si está o no activo, asi que decidimos no seguir de manera estricta la guía proporcionada.

# 9. Ejercicio 8: El ensamble

LLegados a este punto tenemos un scheduler, una MMU, 16 TSSs, pero todavía falta implementar toda la lógica del juego propiamente dicho. Lo que se determina es que hace falta una estructura que mantenga las coordenadas en las que el zombie se encuentra, para asi facilitar el mapeo y la escritura en la pantalla. Ademas hace falta saber a que tipo pertenece cada zombie lanzado, tanto para poder copiar su código de manera efectiva como para poder mostrarlo en pantalla. Hace falta mantener de alguna manera las coordenadas del jugador A y B, los puntajes, la cantidad de zombies disponbiles.

Se determina utilizar $game.c$ para todo ello. Allí se implementará toda la lógica del juego, las posiciones de los jugadores, los puntajes, etc. Dado que $game.c$ posee toda esta información, es lógico pensar que también aquí se maneje la pantalla ya que existe una correspondecia directa entre ambos.

Se crean funciones para que ambos jugadores puedan moverse por la pantalla ($game\_moverJugador()$), cambiar la clase del próximo zombie que será creado ($game\_cambiarClase\_atras()$ y $game\_cambiarClase\_adelante()$) y la funcionalidad de lanzar un zombie (ver $game\_lanzar\_zombi()$). Todas estas funciones estaran mapeadas por interrupciones de teclado a las teclas correspondientes.

Cabe destacar la funcionalidad de $game\_lanzar\_zombi()$ ya que en esta función sucede una parte crítica del juego:

En ella, se buscará primero en una arreglo previamente inicializado, si queda algun zombie ”muerto”del jugador que corresponda, o lo que es lo mismo, existe una tarea para el jugador que corresponda que no este siendo utilizada. De ser el caso, se marca en este arreglo que la tarea ha empezaddo a estar ”viva”(principalente para que esta no pueda ser desplazada por otra tarea), se inicializan las coordenadas X e Y del tablero para el zombie y en base a ella se calculan las direcciones reales que el zombie ocupará.

Una vez que tenemos las direcciones reales del zombie, se procede a inicializar llamar(Inicializar llamar?? esto es asi??(ALE)) a la función $mmu\_inicializar\_zombie$ que como ya hemos dicho, mapea estas posiciones donde corresponde. Asi obtenemos el CR3 del zombie.

Ahora limpiamos su tss, que esta unívocamente ligada al arreglo antes mencionado. Aquí es cuando notamos que no es necesario iniciar antes en $cargarTSS\_zombie()$ las tss de antemano.

Luego cargamos el CR3 del zombie en su TSS con $mapearCr3Tss()$ y asi obtenemos una tss limpia para poder empezar a ejecutar el zombie.

Notamos ademas que este arreglo de si los zombies estan muertos o vivos puede resultar muy util para indicarle al scheduler cual tarea correr, asi que se reimplementa el scheduler para que le pregunte a través de $game\_proximo\_zombie()$ cual será el proximo zombie a ejecutar.