DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming-Zombi Defense

Organización del Computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Christian Chibana	586/13	christian.chiba93@gmail.com
Javier Minces Müller	231/13	javijavi1994@hotmail.com
Nicolás Roulet	587/13	nicoroulet@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

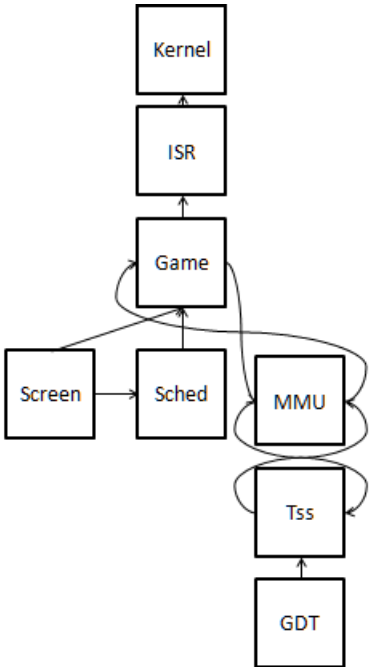
1. Introducción

El objetivo del siguiente trabajo práctico es explorar los conceptos fundamentales de la programación de sistemas, desarrollando para tal fin un sistema con capacidad de manejo de múltiples tareas de manera dinámica y una interfaz de comunicación con el usuario.

La base del sistema consiste en un kernel provisto de estructuras de datos que pe permiten administrar memoria con diferentes niveles de privilegio, capturar eventuales errores en tiempo de ejecución a fin de actuar consecuentemente y recibir interrupciones de dispositivos externos para permitir la entrada de datos. A su vez, está diseñado para poder conmutar tareas a fin de simular una ejecución simultánea.

Estas herramientas permiten implementar un juego de dos jugadores donde cada uno tiene a su disposición zombis (tareas) con los que combatir a su contrincante.

Esta es la estructura de los archivos:



2. Global Descriptor Table (GDT)

La Global Descriptor Table (GDT) es una estructura de datos que permite almacenar descriptores de los diferentes segmentos del sistema. Contiene por convención un descriptor nulo en la primer entrada. En este caso particular, las siguientes 7 entradas se consideran reservadas, por lo que los descriptores utilizados comienzan a partir de la posición 8.

La administración de memoria se basa en cuatro segmentos principales:

- Segmento de código de nivel 0: será utilizado para ejecutar las instrucciones del kernel y demás rutinas que requieran nivel de supervisor.
- Segmento de datos de nivel 0: permite almacenar los datos y las diferentes pilas de nivel de supervisor utilizadas por el kernel y las interrupciones, así como la memoria de video y las páginas adicionales para rutinas de máximo privilegio.
- Segmento de código de nivel 3: es utilizado para ejecutar el código de las tareas, que tienen nivel de privilegio 3.
- Segmento de datos de nivel 3: contiene las pilas de nivel 3 de las tareas.

Estos cuatro segmentos abarcan la misma área de memoria con base en la posicion `0x00000000`, que abarca los primeros 623MB de memoria. A su vez, utilizamos un descriptor para el área de pantalla con la base establecida en la posición `0xB8000` que permite direccionar a los píxeles de la pantalla utilizando su $offset$ relativo al principio de la misma.

Por otro lado, la GDT contiene los descriptores de TSS, que se explican la sección 7.

3. Interruptor Descriptor Table (IDT)

El Interruptor Descriptor Table (IDT) es una estructura de datos que permite almacenar descriptores de las interrupciones internas, de hardware, y de software. Esta tabla es necesaria para atender las interrupciones en modo protegido.

Para inicializar la IDT realizamos una función que, para cada tipo de interrupción, genere el descriptor de la interrupción y se lo asigne a su respectivo índice (el numero de interrupción) dentro de la IDT.

Luego en el archivo $isr.asm$ realizamos la rutina de atención para cada interrupción, imprimiendo el tipo de excepción para aquellas generadas por el procesador, y para las interrupciones del teclado, reloj, y del sistema, se hizo lo correspondiente a lo pedido en el enunciado (se hablara más sobre este tipo de interrupciones en la sección de Interrupciones Externas).

El descriptor de las interrupciones desde el 0 al 19 tienen el mismo selector de segmento y atributos, ya que son todas excepciones del procesador. El selector elegido para estas interrupciones es el de código de nivel 0 (shifteado 3 bits (0x40)) ya que necesita realizar operaciones al nivel de supervisor, por ejemplo desalojar tareas. Sus atributos son (0x8E00): $Present$ 1, DPL de nivel 0 (al igual que es segmento), D en 1 ya que trabajamos en 32 bits, y de $tipo$ Interrupt Gate ($110b$).

La rutina para las excepciones imprime el tipo de excepción en la esquina superior izquierda de la pantalla y llama a la funcion $game\_chau\_zombi$ que marca al zombi que generó la excepción como muerto y desaloja la tarea/zombi, para pasar a la tarea idle. Cargamos la IDT con la función de assembler `lidt [IDT_DESC]`, y habilitamos dichas interrupciones con la instruccion `sti`.

4. Memory Management Unit (MMU)

4.1. Kernel

El kernel debe poder acceder a cada posición de memoria propia, del mapa, y del área libre con nivel de privilegio 0 mediante $identity\ mapping$. En total, $16MB$ de memoria. Para esto, necesita 4 directorios de páginas, cada uno direccionando 1024 páginas de $4KB$.

Los 4 directorios de páginas tienen los mismos atributos de acceso, P en 1 y R/W en 1; el resto de los atributos se encuentran en 0. La base corresponde a la posición de la page table correspondiente a partir de la posición $0x28000$ (cada page table ocupa $4K$).

Para los descriptores dentro de la page table, se utilizaron los mismos atributos (P en 1 y R/W en 1), y se tomó la base desde $0x00000$ aumentando $4K$ por cada descriptor.

Para activar paginación llamamos desde el kernel a las funciones de inicialización, luego cargamos la dirección de la base de la tabla de directorio de páginas en el $CR3$ y habilitamos paginación a partir del bit de PG que se encuentra en el $CR0$.

4.2. Zombis

Las tareas/zombi necesitan dos tablas de páginas. La primera les permite acceder al primer megabyte de memoria, ocupado por el kernel, con nivel de privilegio 3. La segunda les permite tener mapeada la página del mapa en que se encuentra y sus posiciones contiguas.

Para esto creamos una función que mapea una página física a una virtual inicializando las entradas correspondientes de la PD y PT. A su vez, es necesaria una función que realice el proceso inverso, es decir desmapear una página ya mapeada.

Al lanzar un zombi también es necesario copiar el código almacenado en la memoria del kernel a la posición del mapa donde comenzará a ejecutarse.

5. Interrupciones Externas

En este trabajo se implementaron dos rutinas de atención de interrupción para interrupciones externas; una para el reloj y otra para el teclado. Las interrupciones de reloj se producen periódicamente. En cada interrupción, se llama a la función $sched\_proximo\_indice$ y se cambia de tarea. Además, se incrementa un timer, que finaliza el juego si pasa una cantidad de tiempo prudencial sin movimientos o lanzamientos de zombis, declarando ganador al jugador que tenga más puntos.

El teclado produce $scancodes$ al tocar y soltar una tecla, que se leen en el registro `al` mediante la instrucción `in al, 0x60`. Tocar las teclas activas produce una llamada a las funciones del archivo game, $mover\_jugador$, $cambiar\_tipo$ o $lanzar\_zombi$, que reciben como parámetro el jugador y, en los primeros dos casos, la dirección (segun la tecla que se presiona). En el caso de la tecla Y, se explica en la sección "Debugger".

- $mover\_jugador$ imprime y modifica la posición del jugador dentro de las columnas 1 y 79 según la tecla que fue presionada (respetando la forma de mapa circular del enunciado).
- $cambiar\_tipo$ realiza el cambio del tipo de zombi dentro del struct del jugador correspondiente (explicado mas detalladamente en la sección de $Scheduler$), y lo imprime en pantalla representado con la letra correspondiente a su tipo ('M' 'C' o 'G').
- $lanzar\_zombi$ se fija si para el jugador que presionó la tecla para lanzar, que los zombis que le quedan para lanzar (parte del struct $info_jug$) sea mayor a 0 y los zombis en pantalla menor a 8 (tambien parte del struct), si se cumple esta condición, se busca una posición libre en el array de tareas zombis de ese jugador para completar la información correspondiente a ese zombi (struct $info_zombi$) e inicializar el zombi (se utiliza la funcion $tss_completar_libre(tss*t, chartipo, charjugador, intpos)$ que llama a la función $mmu_inicializar_dir_zombi$ y completa toda la tss).

5.1. Syscalls

Los zombis pueden solicitar una interrupción (0x66) con nivel de privilegio 3 (ya que no queremos que una tarea zombi pueda ejecutar operaciones del supervisor) para moverse a una posición adyacente. Cuando esto ocurre, es necesario cambiar su posición, remapear las páginas del mapa a las que tiene acceso y copiar su código a la nueva posición. Si llega a un límite, muere, sumando un punto para el jugador que se encuentra del lado opuesto a ese límite.

6. Task Segment Selector (TSS)

El procesador x86 en modo legacy 32 permite ejecutar cambios de tarea por hardware. Para esto es necesario conservar el contexto de las tareas en una estructura determinada. Esta estructura es un segmento de memoria especial, con los bits de tipo identificándola como tal e indicando si esta ocupada.

El procesador reconoce esta estructura al ejecutar un `jmp far selector:offset` y hace el cambio de tarea de la siguiente forma:

- Preserva los registros de la tarea actual en el TSS correspondiente. El `eip` guardado será el correspondiente a la instrucción siguiente al `jmp far`.
- Busca la nueva TSS utilizando el selector pasado como parámetro y carga los registros de la nueva tarea. Ahora está en condiciones de comenzar a ejecutar la nueva tarea.

Durante ésta operación, el procesador verifica que concuerden los niveles de privilegio de selectores y segmentos y que dichos segmentos sean válidos y se encuentren presentes. En caso contrario, saltará una excepción.

En este trabajo se declararon 18 entradas en la GDT cada una como descriptor de TSS: una para cada tarea zombi (8 para cada jugador) con $DPL = 3$, una para la $tarea\ inicial$, y otra para la $idle$ (la tarea de nivel de kernel encargada de ocupar los espacios de tiempo en los que no haya otras tareas ejecutándose). La base de éstos segmentos será inicializada posteriormente al declararse las TSS (para las tareas zombis,

pasando como base la dirección de la posición dentro del array de TSS zombis correspondiente a cada zombi).

La tarea inicial tiene como único objetivo tener un contexto antes de ejecutar el primer `jmp far` para comenzar la ejecución por la tarea idle.

7. Scheduler

El scheduler indica al procesador qué tarea ejecutar por cada interrupción de reloj. Para esto, usamos una estructura $info\_zombi$, con la información relevante de cada tarea, incluyendo un bit que indica si la tarea/zombi está viva.

Las tareas se encuentran en dos arrays de ocho posiciones cada uno, uno para cada jugador, inicializado con tareas muertas. Al llamar a la función (esa función), busca en el jugador contrario al último que jugó una tarea $viva$, dejando su última tarea ejecutada para el final. La función devuelve su índice en la GDT, o el índice de la tarea IDLE si no hay tareas vivas.

Esto permite ejecutar el `jmp far` a la posición devuelta. En caso de que haya un solo zombi vivo (o ninguno) no se realiza este salto, porque no se puede saltar de una tarea a sí misma.

También se guarda la información de los jugadores: cuántos zombis tienen por lanzar, cuántos puntos tienen, y su posición y tipo de zombi a lanzar.

8. Debugger

Para implementar el debugger creamos dos flags: uno indica si se está en modo debug, la otra, si se está mostrando la información de una tarea desalojada por una excepción.

Tocar la Y cuando el flag de $mostrando$ está en 0 cambia el valor del flag de $debug$. Si este está en 1 y una tarea genera una excepción, se imprime su estado en pantalla.

Para esto, creamos una función en C. Antes de llamar a esta función, se guardan en la pila todos los valores a imprimir. Luego, se hace `push esp`, que es el único parámetro que toma la función. A partir de este, puede ir recorriendo la pila y devolviendo los valores encontrados.

esp →	dirección de retorno	
	esp	(valor al momento llamar a la función)
	eax	
	...	
	cr0	(primer registro pusheado antes de llamar a la función)
	stack	(stack antes de pushear valores)

La pantalla se guarda completa a partir de la posición `0xB9FA0`, y es recuperada al presionar la tecla Y. Mientras el debugger se está mostrando, las interrupciones entran en una rutina separada que solo habilita la que corresponde a tocar esta tecla.

9. Bonus Track

9.1. Zombis inteligentes (wait...)

Como *bonus track* se implementó una versión mejorada del zombi clérigo (`tareaAC.c`) que avanza realizando bucles y en cada turno copia a las posiciones aledañas del mapa código basura que produce que cualquier zombi que se encuentre en ellas muera al llegar su turno e intentar ejecutar ese código.

9.2. Videojuego Favorito

Hasta la fecha, el puesto al mejor videojuego de zombis se lo lleva indudablemente el TP-Orga-2 desarrollado por Chibana-Minces-Roulet Inc.

10. Bibliografía

- Apuntes de clase Práctica y Teórica
- Manuales de Intel