Organización del Computador II

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

23 de junio de 2017

Trabajo Práctico Número 3

Zombi Defence

Grupo: "Dora La Consola"

Integrante	LU	Correo electrónico
Reyes Mesarra, Darío René	838/15	dario6reyes6@yahoo.com.ar
Pazos Méndez, Nicolás Javier	709/15	npazosmendez@gmail.com
Balbi, Pablo Luis	707/15	pablo.l.balbi@gmail.com

Cómo entender nuestro kernel y no morir en el intento - Vol I

Índice

1. Inicialización

Compuesto por kernel.asm

`kernel.asm` es el punto de entrada al código del kernel. Se encarga de realizar las operaciones de más bajo nivel del mismo, asegurándose de cargar los valores correspondientes en los registros internos para que los mecanismos del procesador (paginación, interrupciones, modo protegido, tareas, etc) funcionen correctamente. También hace una delegación general de trabajo a los demás módulos del kernel para inicializar las estructuras de datos que hagan falta. En resúmen:

- Configurar video
- Cargar la GDT y saltar a modo protegido
- Llamar a las rutinas que inicializan la MMU
- Cargar un mapa de memoria con identity mapping de los primeros 4MB y activar paginación
- Llamar a las rutinas que inicializan las TSS de todas las tareas, y cargar un TR válido al contexto actual
- Llamar a las rutinas que inicializan la IDT y cargarlas
- Configurar el pic para remapear interrupciones externas y activar interrupciones

Al activarse las interrupciones, se corre por primera vez la rutina de atención del clock, que contiene el código del scheduler. Llegados a este punto, todo se encuentra inicializado. El control pasa al scheduler, las tareas, y las interrupciones generadas por excepciones, o por eventos generados por el usuario.

2. Segmentación

Compuesto por gdt.c, gdt.h

En `gdt.c` se encuentran **definidos** los descriptores de segmentos principales: tanto de código y de datos de nivel 0 (kernel y tarea idle) y 3 (tareas zombis). Como tomamos el modelo flat como estrategia de segmentación, todos los segmentos comparten base 0 y límite $623MB$. El problema de la protección del kernel entonces no será resuelto por segmentación, sino por paginación. De cualquier forma, es importante que el nivel de privilegio del segmento de código sea 3 para las tareas, pues es lo que define el CPL (Current Privilege Level) de las tareas, del cual depende el checkeo de privilegios de paginación.

No tiene interfaz alguna, pues la información adentro de la GDT no se vuelve a tocar, y para acceder a la misma sólo hacen falta los selectores correctos, que están definidos como macros en `defines.h`.

La GDT también contiene las task-gates de las tareas, pero las mismas serán descriptas en la sección del mismo nombre.

3. Paginación

Compuesto por kernel.asm, mmu.c y mmu.h

Generalidades

Este módulo se encarga de todas las rutinas relacionadas con paginación.

Las páginas físicas que vamos a utilizar van a provenir de un *área libre* en memoria definida en el rango $0x100000 - 0x3fffff$. Vamos a inicializar una variable *prox_pag_lib* al comienzo del área libre que apuntará en todo momento a una página libre que podamos utilizar. Al pedir memoria, lo único que haremos será apuntar a la siguiente página, sin preocuparnos por liberar memoria (dado que sólo vamos a necesitar memoria para 40 tareas zombi como máximo, no va a ser ningún problema).

Se hará un mapeo de memoria de los primeros 4MB con identity mapping para el kernel y la tarea idle como pide el enunciado.

Para las tareas zombis, se mapearán los primeros 4MB con identity mapping y DPL supervisor, y las 9 páginas de la posición del mapa en donde se encuentra el zombi a $0x8000000 - 0x8008fff$ con DPL usuario. Esto implica que las tareas normalmente sólo tendrán acceso a las direcciones lineales en $0x8000000 - 0x8008fff$ (con permisos de lectura y escritura sobre posiciones en el mapa). Al producirse la interrupción del scheduler en una tarea, como el segmento de código cambia a uno de nivel 0 (definido en la IDT), las rutinas tienen acceso a las estructuras del kernel.

Inicialización

Hay detalles a explicar en cómo fueron inicializadas los esquemas de memoria de las tareas. Para no desaprovechar memoria, decidimos no pedir memoria para cada zombi nuevo creado. Lo que hicimos fue pedir memoria para las 16 posibles tareas desde un principio. Dicha inicialización la comienza `tss.c`, que es la que termina guardando las direcciones de memoria de las páginas pedidas.

La función que se llama para esto es `mmu_inicializar_esquema_zombi`. Toma como entrada un *jugador* y un *y* que indica dónde está el zombi verticalmente (el x está implicito en el jugador, ya que todos los zombis comienzan siempre en las mismas columas). Lo que hace la función es pedir dos páginas, una para un page directory, y otra para un page table. Para implementar el identity mapping de los primeros 4MB, aprovecha la tabla de páginas del kernel en la dirección $0x28000$ que ya utiliza el esquema de paginación del kernel. La página nueva que pedimos para una page table la usamos para mapear a las direcciones $0x800000 - 0x8008fff$.

Para completar la page table, se delega a la rutina `mmu_mapear_vision_zombi`, que a su vez aplica `mmu_mapear_pagina` (del enunciado) nueve veces para mapear individualmente las páginas lineales a partir de la $0x8000000$ a las posiciones adyacentes al zombi. La función toma *jugador*, para saber en qué orden mapear las páginas (que cambian según el jugador, de tal forma que el zombi es agnóstico de a qué bando pertenece), un *x*, un *y*, con el cual calcularemos las direcciones físicas del mapa a las que mapearemos y un *cr3* que indicará sobre qué mapa de memoria se aplicará el mapeo. Observar que esta instrucción no inicializa nada, solo mapea las direcciones que componen la 'visión' del zombi, con lo cual es una función bastante general. De hecho, la usaremos también para la syscall mover.

4. Tareas

Compuesto por tss.h y tss.c

La mayor responsabilidad del archivo `tss.c` es la de inicializar las estructuras de las tareas (la función `tss_inicializar`). Por cada tarea hay una variable global con su TSS (para ser más precisos, las tareas zombis están guardadas en arrays globales). Para inicializar una tarea se completa su TSS con los valores que corresponden, y se le asigna una task-gate en la GDT.

Como mencionamos anteriormente, la idea es inicializar todo lo que necesitamos para los zombis en un principio, y reutilizar esos recursos. `tss_inicializar_zombi` toma un jugador y un número de tarea (del 0 al 7), y:

- Pide memoria para el cr3 (con `mmu_inicializar_esquema_zombi`) y el esp0 (con `mmu_prox_pag_libre`)
- Completa la TSS de la tarea (utilizando el cr3 y el esp0)
- Agrega la entrada de la GDT (utilizando la dirección de la tss de la tarea)

Una vez hecho esto para todos los zombis, los `esp0` y `cr3` se guardan como variables globales. Esto es para sobrescribir la TSS con los valores iniciales cuando querramos utilizarlo para un zombi nuevo. Esto ocurrirá en la rutina de atención del teclado, en la que al apretar una tecla Shift se lanzará un zombi, y por ende, una nueva tarea. Dicha rutina necesitará acudir a `tss.c`, llamando a la función `tss_refrescar_zombi`, que toma un jugador y un número de task, y utilizando el cr3 y el esp0 que le corresponden, completa la tss con sus valores iniciales.

Se dispone de las funciones `tss_leer_cr3` y `tss_escribir_cr3` para modificaciones que querramos hacer en el mapa de memoria de un TSS determinado.

5. Interrupciones

Compuesto por idt.h, idt.c, isr.asm, interrupt.c y interrupt.h

Para el archivo `idt.c` no hubo muchas decisiones de diseño que tomar: se encuentra allí la rutina que inicializa las entradas de la IDT utilizando la macro proporcionada por la cátedra. En `isr.asm` están definidas todas las rutinas de atención e interrupción: las excepciones arquitecturales, el teclado, el clock, y la syscall mover. Suelen llamar a rutinas de C ubicadas en `interrupt.c` para programar con más comodidad.

El clock tiene el código del scheduler, con lo cuál tendrá su propio apartado. Por la manera en la que organizamos el código, la *isr* del syscall mover terminó teniendo muchas responsabilidades, con lo cuál también tendrá una sección aparte.

Excepciones

En general, todas las excepciones se manejan de una forma parecida. Primero nos encargamos de guardar el contexto para una posible llamada a Debug (más información en dicha sección). Después, hay dos variantes: o la excepción fue generada por una tarea zombi, o por el kernel/tarea idle. Para verificar en qué caso estamos, utilizamos el CPL del proceso interrumpido, que se encuentra en el CS guardado en la pila.

En el primer caso, cuando venimos de una tarea, como pide el enunciado, se 'mata' la tarea llamando a `handle_zombi_exception` de `interrupt.c`. Lo que hace esta rutina es avisarle al scheduler que el zombi ya no existe, y pintar la pantalla para notificar al usuario. El número de 'restantes' no baja en esta instancia, sino que baja al lanzar un zombi, asi que no hace falta modificar el estado global del juego. Una vez terminado esto, hacemos un task-switch a Idle. Nunca vamos a volver a la tarea, por lo que no hace falta completar con más código. Esto ocurre porque el scheduler dio de baja el contexto de la tarea, por lo que se lo saca de la rotación de tareas. Cuando el scheduler quiere reutilizar el 'slot' del zombi, la TSS asociada al mismo es reescrita, momento en el cual perdemos el contexto viejo definitivamente.

El segundo caso, en el que venimos del kernel, marca fallo no planificado. Por dicha razón, decidimos en `handle_kernel_exception` escribir en pantalla datos de la excepción generada (nombre, número de vector, registros varios), y haltear el procesador.

Ambos casos son muy parecidos, pero hay un detalle que complejizó el código. Ese detalle es que algunas excepciones pushean el Error Code y otras no; esto hace que la lectura de estos datos

no pueda hacerse igual para todas ellas. Lo resolvimos pusheando un valor cualquiera para las excepciones que no generan Error Code, de tal manera que a partir de ese entonces podamos utilizar el mismo código para cualquier caso (esto es posible porque no hacemos uso del Error Code).

Teclado

La rutina de atención e interrupción lee inmediatamente el dato proveido por el teclado, y se lo pasa a la rutina `handle_keyboard` de `interrupt.c`. Ésta verifica qué tecla se pulsó, y llama a las rutinas de `game.c` correspondientes. Por ejemplo, al presionar `W` se llama a `game_mover_jugador` (pasándole la dirección y el jugador que se movió). También se puede prender y apagar el modo de debug con la tecla Y, por lo que esta rutina también lo verifica (y llama a funciones de `debug.c` para que se encarguen).

6. Juego

Compuesto por game.c y game.h

Este módulo contiene las variables que mantienen el estado del juego. Ellas son:

- Las posiciones y de los cursores de los jugadores
- Los tipos de zombi actuales que tienen seleccionados los jugadores
- Los zombis restantes de los jugadores
- Los puntajes de los jugadores
- Las posiciones de los zombis activos
- Los tipos de los zombis activos
- Si el modo debug está activo
- Si el debugger ya detectó una excepción

Las últimas dos se verán en Debug. El resto son variables que tienen lo que necesitamos para la lógica del juego. Siempre que hay un cambio en el juego, como la posición de un zombi, de un jugador, o que se mate o lance un zombi, estas variables deben modificarse para reflejar el estado en el que se encuentra el juego. Por comodidad, están declaradas en `game.h` y se utilizan directamente en archivos como `interrupt.c` o `screen.c`, dependiendo del evento.

Además, este módulo provee rutinas que representan las diversas acciones dentro del juego. Están íntimamente relacionadas con la *isr* del teclado, que es de donde estas funciones son llamadas dependiendo de qué teclas se presionaron.

La función más interesante es `game_lanzar_zombi`, que crea una nueva tarea. Primero, se le pregunta al Scheduler si hay un lugar libre (`sched_hay_lugar_zombi`), y si lo hay, se le pide el mismo (`sched_indice_libre`), además de avisarle que a partir de ahora está activo `sched_activar_zombi`. Se pide el cr3 del TSS asociado a dicho índice (`tss_leer_zombi`), y se pisa el viejo mapeo de las páginas que representan la visión del zombi ($0x8000000 - 0x8008fff$) con un mapeo hacia la posición en la que zombi está siendo inicializado (`mmu_mapear_vision_zombi`). Además, también se resetea el contexto entero de la tarea con `tss_refrescar_zombi`). Finalmente, se actualizan las variables del juego, y se dibuja en pantalla los cambios realizados.

7. Scheduler

Compuesto por isr.asm, sched.c y sched.h

La rutina principal del scheduler en `isr.asm` depende de `sched_proximo_indice`, que le indica al selector la próxima tarea a correr. Si no se cargó ninguna tarea todavia ($TR = 0$) o si la próxima tarea es la misma que la actual (porque hay una sóla tarea corriendo), entonces no se hace nada (ya que de llevarse a cabo un Task-Switch se produciría caos, fuego y destrucción), retornando de la interrupción. De lo contrario, hacemos un Task-Switch a la nueva tarea mediante un jump far con un selector al task-gate indicado (el 'próximo índice' que se obtuvo. El RPL de dicho selector debe ser 0, porque es el privilegio que necesitamos para iniciar la conmutación de tareas (el CPL también es 0, porque la interrupción del reloj está en un segmento de nivel 0). Un comportamiento especial que posee el scheduler, consiste en que si se encuentra prendido el flag de *debug*, entonces `sched_proximo_indice` siempre devoldera el índice de la tarea Idle, *pausando* el juego.

Si bien la isr del clock se encarga de iniciar las conmutaciones de tareas, es en el archivo `sched.c` en el que se encuentran definidas las estructuras que utiliza el scheduler para llevar a cabo el seguimiento de qué tareas estan corriendo. Son 8 las tareas que pueden estar corriendo simultaneamente para cada jugador, y vamos a administrar las tareas pensando que tenemos 8 slots en donde podemos ubicar nuestras tareas activas. Para implementar esto utilizamos dos arreglos de booleanos de longitud 8 (*tasks_A* y *tasks_B*) , en donde un 1 marca que el 'slot' está siendo utilizado por un zombi activo.

Para ejecutar las tareas activas de la forma que requiere el enunciado guardamos la última tarea ejectuada de cada jugador (*running_A* y *running_B*), y qué jugador es el próximo al que el scheduler le debe transferir el control en el próximo quantum (*next_player*).

El archivo también contiene funciones de interfaz con las estructuras del mismo (para indicar qué tarea esta siendo ejecutada, para agregar nuevas tareas al scheduler, o para consultar las mismas). Resulta conveniente tener estas funciones porque, por ejemplo, el scheduler es el único que tiene guardado quién es el último zombi que se acabó de ejecutar.

8. Debug

Compuesto por debug.c y debug.h

Estos archivos fuente contienen dos conjuntos de funciones: un primer conjunto de funciones que sirve para imprimir en pantalla errores varios (por ejemplo excepciones del kernel sin resolver), y otro conjunto que compone el funcionamiento del modo debug del juego, que pide la consigna. Las primeras funciones son escencialmente renombres a llamadas de la función `print` (y sus variantes) con determinados valores de posición y atributos.

Nótese que el modo debug tiene su utilidad para excepciones de los zombis, y no así las del kernel. Las excepciones del kernel no se manejan de la misma manera (ver Interrupciones), y no dan lugar a este modo.

Al presionarse la tecla Y se activa el modo debug llamando a la función `debug_on`, que lo primero que hace es poner en *true* la variable global del juego `debug`, indicando que se prendió el modo debug. Este valor de verdad hace que el scheduler no actue, pausando todo movimiento del juego instantáneamente. Luego se procede a guardar los valores de la pantalla que se sobreescribirán al mostrar la ventana; para esto se llama a la función `backup_screen`, que copia los datos correspondientes a un arreglo global, al que accederá también la función `restore_screen` al salir del modo debug, que lleva a cabo el proceso inverso. Este arreglo está definido en el mismo archivo, por lo que se encuentra en el área de memoria del kernel.

Luego de haber hecho el *backup* de la pantalla correspondiente, se procede a imprimir la información guardada correspondiente al contexto que provocó la primera excepción. Esta se encuentra

almacenada en un *struct* global, cuyos campos son todos los registros de interés. Esta información es guardada por la función `debug_save_context`, que es llamada por una excepción únicamente si fue la primera en ocurrir; para ello se tiene otro booleano global del juego, `debug_hubo_excepcion`, que comienza en falso y se establece en verdadero cuando ocurre la primera excepción.

Los valores de la pila del contexto que guarda el modo debug merecen una mención especial. No podemos asegurar que el registro `esp` esté apuntando a lo que la rutina utilizaba como pila, pero lo asumiremos así. Sin embargo, tomaremos la precaución de verificar que la dirección de memoria que contenga se encuentre en el rango $0x8000000 - 0x8008fff$. Si es así, se muestran tantos valores de 4 *bytes* como indique la distancia entre `esp` y `ebp`, con un tope máximo de 10. Si no se encuentra en rango, el modo debug no muestra valores de la pila.

Si se presiona la tecla Y y el modo debug está prendido, entonces se llama a la función `debug_off`, que simplemente restaura la pantalla y establece al booleano `debug` en *false*, para que el scheduler retome su funcionamiento normal.

9. Syscall

Compuesto por syscall.h, isr.asm y interrupt.c

Los llamados a la syscall se realizan mediante la funcion syscall_mover definida en syscall.h (proveída por la cátedra). Como la syscall es, en otras palabras, una llamada a un servicio del kernel, la comunicación entre la tarea llamadora y el kernel se realiza por medio de una interrupción (número 102). Una vez realizada la interrupción, la misma es atendida por su handler, definido en isr.asm. El mismo realiza una llamada a `handle_syscall_mover`, pasándole como parámtro la dirección a la cual mover la tarea (zombi).

En `interrupt.c`, se enucentra definida la rutina (`handle_syscall_mover`) que lleva a cabo la acción de la syscall (mover la tarea en una dirección dada). La misma consiste en:

1. Identificar la tarea que realizo el llamado a la syscall.
2. Calcular la posición del mapa destino, a la cual se moverá la tarea (zombie).
3. Replicar el código de la tarea, en la posición destino.
4. Mapear en el esquema de paginación de la tarea la nueva posicón del código, así como sus páginas adyancentes (la cuales son la zona visible por la misma).
5. Pintar en la pantalla al zombie en su nueva locación, y dejar un *trace* en su anterior.
6. Si la tarea llego al otro extremo del mapa, actualizar los puntajes, llamar a la rutina encargada de destruir la tarea, y en caso de algún jugador haya ganado, festejarselo apropiadamente.

La información que se recupera para identificar la tarea llamadora, se obtiene mediante consultas a las estructuras del juego, asi como las del scheduler. Por otro lado, las rutinas utilizadas para replicar a la tarea, e interactuar con su esquema de paginación, son las mismas que se utilizaron (levemente modificadas en el caso de la replicación, ya que se debe copiar también el stack de la tarea) al inicializarlas, vistas en la sección de paginación. En el caso de que el zombie haya ganado un punto para su respectivo jugador, los mismos se actualizan en variables globales, pertenecientes a la estructura del juego.