

Trabajo Práctico 3

System Programming - Zombi defense

Organización del Computador 2

Primer Cuatrimestre 2017

1. Objetivo

Este trabajo práctico consiste en un conjunto de ejercicios en los que se aplican de forma gradual, los conceptos de *System Programming* vistos en las clases teóricas y prácticas.

Se busca construir un sistema mínimo que permita correr 16 tareas concurrentemente a nivel de usuario. El sistema será capaz de capturar cualquier problema que puedan generar las tareas y tomar las acciones necesarias para quitar a la tarea del sistema. Además, estas tareas podrán ser cargadas en el sistema dinámicamente por medio del uso del teclado.

Los ejercicios de este trabajo práctico proponen utilizar los mecanismos que posee el procesador para la programación desde el punto de vista del sistema operativo enfocados en dos aspectos: el sistema de protección y la ejecución concurrente de tareas.

2. Introducción

Para este trabajo se utilizará como entorno de pruebas el programa *Bochs*. El mismo permite simular una computadora IBM-PC compatible desde el inicio, y realizar tareas de debugging. Todo el código provisto para la realización del presente trabajo está ideado para correr en *Bochs* de forma sencilla.

Una computadora al iniciar comienza con la ejecución del POST y el BIOS, el cual se encarga de reconocer el primer dispositivo de booteo. En este caso dispondremos de un *Floppy Disk* como dispositivo de booteo. En el primer sector de dicho *floppy*, se almacena el *boot-sector*. El BIOS se encarga de copiar a memoria 512 bytes del sector, a partir de la dirección 0x7c00. Luego, se comienza a ejecutar el código a partir esta dirección. El boot-sector debe encontrar en el *floppy* el archivo `kernel.bin` y copiarlo a memoria. Éste se copia a partir de la dirección 0x1200, y luego se ejecuta a partir de esa misma dirección. En la figura 1 se presenta el mapa de organización de la memoria utilizada por el *kernel*.

Es importante tener en cuenta que el código del *boot-sector* se encarga exclusivamente de copiar el *kernel* y dar el control al mismo, es decir, no cambia el modo del procesador. El código del *boot-sector*, como así todo el esquema de trabajo para armar el *kernel* y correr tareas, es provisto por la cátedra.

Los archivos a utilizar como punto de partida para este trabajo práctico son los siguientes:

- `Makefile` - encargado de compilar y generar el *floppy disk*.
- `bochsrc` y `bochsdbg` - configuración para inicializar *Bochs*.

- `diskette.img` - la imagen del *floppy* que contiene el *boot-sector* preparado para cargar el *kernel*. (*viene comprimida, la deben descomprimir*)
- `kernel.asm` - esquema básico del código para el *kernel*.
- `defines.h` y `colors.h` - constantes y definiciones
- `gdt.h` y `gdt.c` - definición de la tabla de descriptores globales.
- `tss.h` y `tss.c` - definición de entradas de TSS.
- `idt.h` y `idt.c` - entradas para la IDT y funciones asociadas como `idt_inicializar` para completar entradas en la IDT.
- `isr.h` y `isr.asm` - definiciones de las rutinas para atender interrupciones (*Interrupt Service Routines*)
- `sched.h` y `sched.c` - rutinas asociadas al *scheduler*.
- `mmu.h` y `mmu.c` - rutinas asociadas a la administración de memoria.
- `screen.h` y `screen.c` - rutinas para pintar la pantalla.
- `a20.asm` - rutinas para habilitar y deshabilitar A20.
- `imprimir.mac` - macros útiles para imprimir por pantalla y transformar valores.
- `idle.asm` - código de la tarea *Idle*.
- `game.h` y `game.c` - implementación de los llamados al sistema y lógica del juego.
- `syscalls.h` - interfaz utilizar en C los llamados al sistema.
- `tarea1.c` a `tarea8.c` - código de las tareas (*dummy*).
- `i386.h` - funciones auxiliares para utilizar *assembly* desde C.
- `pic.c` y `pic.h` - funciones `habilitar_pic`, `deshabilitar_pic`, `fin_intr_pic1` y `reseteo_pic`.

Todos los archivos provistos por la cátedra **pueden y deben** ser modificados. Los mismos sirven como guía del trabajo y están armados de esa forma, es decir, que antes de utilizar cualquier parte del código **deben** entenderla y modificarla para que cumpla con las especificaciones de su propia implementación.

A continuación se da paso al enunciado, se recomienda leerlo en su totalidad antes de comenzar con los ejercicios. El núcleo de los ejercicios será realizado en clase, dejando cuestiones cosméticas y de informe para el hogar.

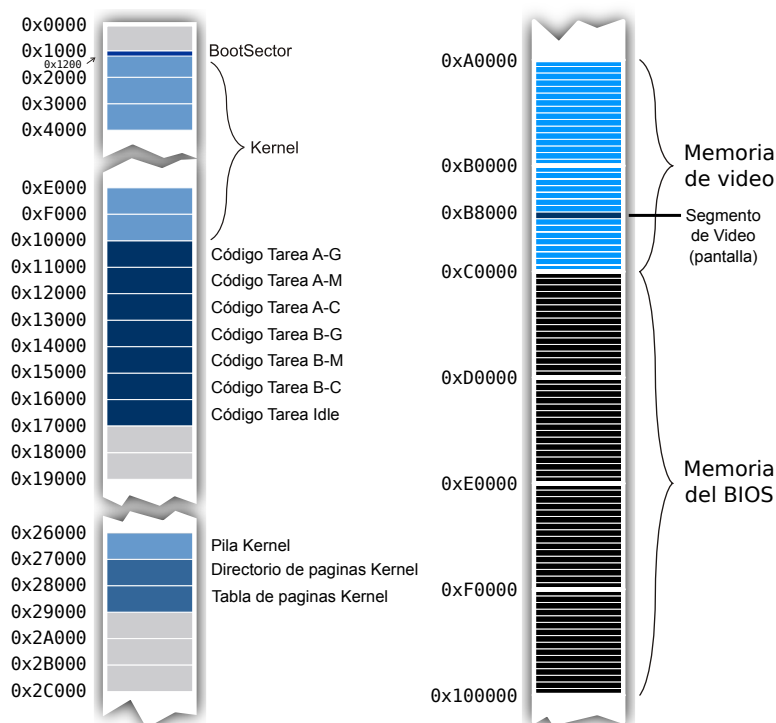


Figura 1: Mapa de la organización de la memoria física del *kernel*

3. Zombi defense

El concepto de *Zombi* se puede extender a diferentes contextos, en la vida real nos encontramos con documentos como “CONPLAN 8888” que intentan clasificar y preparar a la milicia para la defensa contra un posible ataque zombi. En computación el concepto de zombi es un tanto diferente, esta relacionado a cuando un agente externo toma el control de una computadora. Esto convierte al ordenador en un equipo zombi. Extendiendo esta idea a procesos, vamos a definir como zombi a un proceso que busca modificar el código de otros procesos para tomar el control, es decir “comerse su cerebro”.

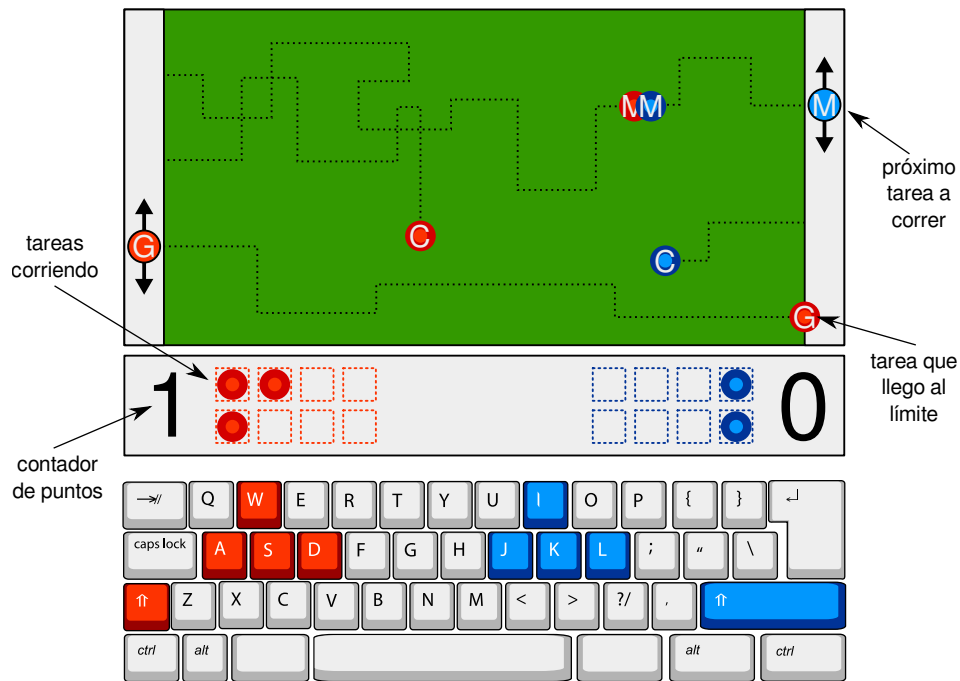


Figura 2: El Juego

En este trabajo práctico vamos a implementar un juego de dos jugadores (A y B). Cada jugador tendrá la posibilidad de enviar zombis contra su contrincante. Estos zombis serán tareas del sistema que se moverán por la memoria en búsqueda de otras tareas para comer sus cerebros.

Los dos jugadores podrán tener un máximo de 8 zombis en juego de tres tipos diferentes, Zombi Guerrero, Zombi Mago y Zombi Clerigo. Si un zombi “muere” (tecnicamente es destruido, porque un zombi no puede morir), entonces libera el slot y el jugador puede lanzar un nuevo zombi. Los zombis son tareas con una funcionalidad, moverse. Cuando un zombi decide moverse, llama al sistema que mueve toda su memoria a una nueva posición. En la figura 5 se da cuenta del único llamado al sistema que puede hacer el zombi. Los zombis, como es conocido, no tienen un cerebro en pleno funcionamiento, por lo que no saben donde están de la pantalla. Estos pueden moverse en cualquier dirección siendo su objetivo llegar al lado contrario de la pantalla, rica en jugosos cerebros.

Cuando un zombi llega al lado contrario de la pantalla muere instantaneamente y se cuenta como un punto para el jugador contrario de donde llego el zombi. Es decir, que si un zombi

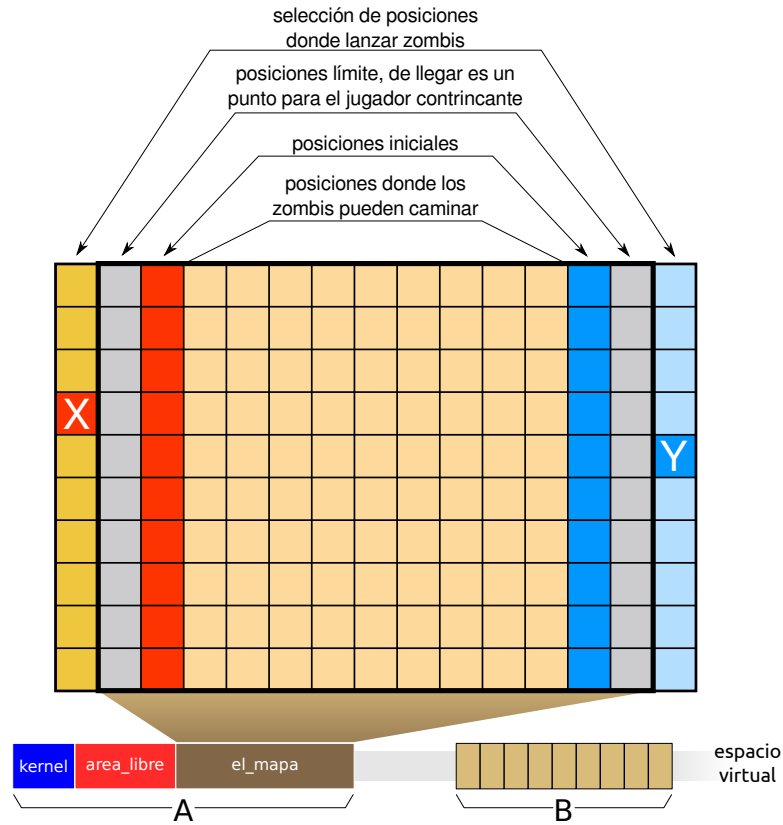


Figura 3: Areas del mapa (ejemplo fuera de escala)

se mueve en sentido contrario al objetivo, esto se cuenta como un punto en contra. Cada jugador puede lanzar como máximo 20 zombis en total. El juego termina cuando a ninguno de los jugadores les quedan mas zombis que lanzar o la cantidad de puntos de alguno de los dos jugadores alcanza los 10 puntos. Gana el jugador obtuvo mas puntos.

Es posible llegar al caso en que todos los slots estén ocupados y no se puedan lanzar mas zombis. En este caso se da por terminado el juego luego de un tiempo prudencial en que no pasa nada (no hay cambios ni movimientos de ningún zombi).

3.1. Zombis (Tareas)

El sistema correrá 16 tareas concurrentemente. Las mismas podrán realizar una sola acción “moverse”, que será implementada como un servicio del sistema.

Cada zombi o tarea tendrá asignada inicialmente una sola página de memoria de 4kb, esta pagina estará ubicada dentro del área de memoria denominada *el_mapa*. Este corresponderá a un área de memoria física de 78×44 paginas de 4kb.

Inicialmente los zombis comenzarán en uno de los extremos del mapa y se moverán hacia el otro extremo siguiendo el camino que determine su código. En la figura 3 se pueden ver las posiciones que puede tomar un zombi en el mapa. Los extremos superior e inferior del mapa son ciclicos, es decir que si se llega a uno de los lados se continua en el otro como muestra la figura 4

Las tareas zombi tendrán en todo momento mapeado a su área virtual todas las paginas alrededor de la posición donde se encuentre su código. Si un zombi decide moverse, se copiará su código a la página donde se mueva y se remapearan todas sus paginas alrededor de esta.

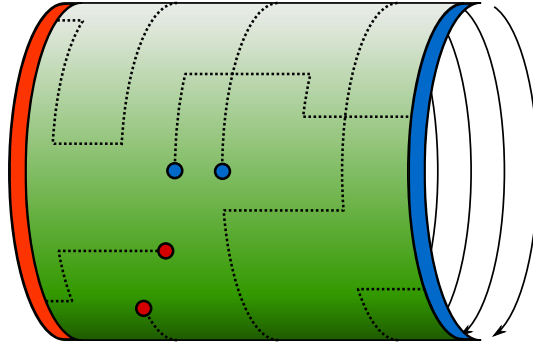


Figura 4: Mapa circular

El orden de mapeo es el indicado en la figura 5.

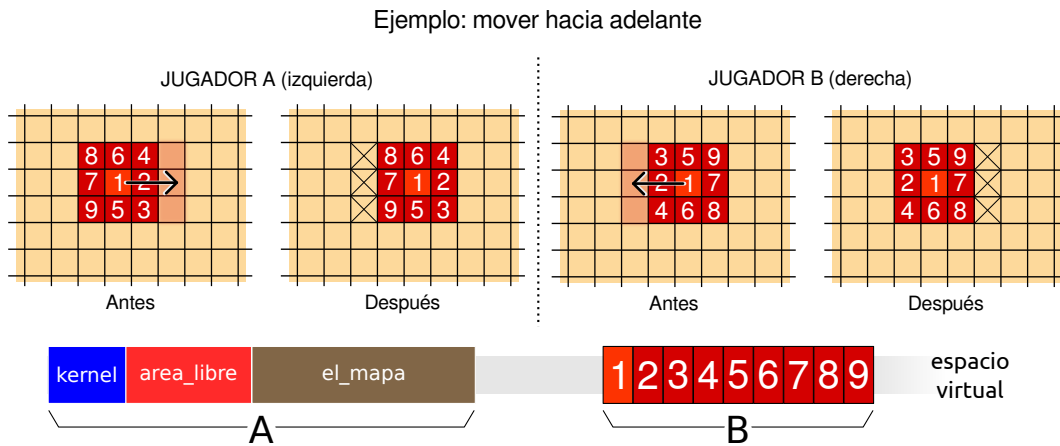


Figura 5: El Juego

3.1.1. A mover zombis (Servicios del Sistema)

El sistema provee un único servicio que corresponde a mover zombis. Este servicio será implementado como una *system call* mapeada a la interrupción 0x66. Su único parámetro será pasado por medio del registro EAX que indicará la dirección de desplazamiento del zombi.

A continuación se presenta la tabla para la codificación de desplazamientos:

Adelante	=	0x83D
Atras	=	0x732
Derecha	=	0x441
Izquierda	=	0xAAA

Tener en cuenta que los desplazamientos son relativos al jugador. Por ejemplo, para el jugador de la derecha, un desplazamiento hacia **Adelante** sería en dirección contraria que para el jugador de la izquierda. Lo mismo sucede si consideramos el desplazamiento a **Derecha**, para el jugador de la derecha será mover al zombi hacia la posición superior en la pantalla, mientras que para el jugador izquierda será mover al zombi hacia la posición inferior en la

pantalla. Recordar que los movimientos a Derecha y a Izquierda, son ciclicos en el mapa. (figura 4)

Por ultimo, es fundamental tener en cuenta que una vez llamado el servicio, el *scheduler* se encargará de desalojar a la tarea que lo llamó para dar paso a la próxima tarea. Este mecanismo será detallado mas adelante.

3.1.2. Nuevo Zombi y lógica de juego

Una funcionalidad que se debe destacar es la posibilidad que brinda el sistema de lanzar nuevos zombis. Esta parte del sistema será implementada en nivel supervisor y por lo tanto no podrá ser afectada por las tareas zombis. La lógica del juego corresponde a solucionar los siguientes problemas:

- Lanzar un nuevo zombi:
Esta funcionalidad será resuelta a nivel del scheduler que generará una nueva tarea. El llamado será desde la interrupción de teclado.
- Mover a los jugadores y controlar teclas:
Esta funcionalidad será resuelta en la interrupción de teclado llamando a funciones implementadas para la lógica del juego.

Las teclas que utilizarán cada uno de los jugadores se presentan en la siguiente tabla:

Acción	Jugador A (izquierda)	Jugador B (derecha)	Descripción
↑	w	i	Mover jugador abajo
↓	s	k	Mover jugador arriba
→	d	l	Cambiar tipo de Zombi (+)
←	a	j	Cambiar tipo de Zombi (-)
↑↑	LShift	RShift	Lanzar Zombi

Como fue mencionado, los tipos de zombi son tres, el orden relativo entre estos es: Guerrero, Mago, Clerigo. Cada uno corresponde a un tipo de tarea distinta que pueden lanzar los jugadores. Cuando se preciona la tecla ↑↑ correspondiente se lanza el tipo seleccionado de zombi.

3.1.3. Organización de la memoria

Cada uno de los zombis tiene mapeadas las áreas de *kernel* y *libre* con *identity mapping* en nivel 0. Sin embargo, área de *mapa* no está mapeada. Esto obliga al *kernel* a mapear el *mapa* cada vez que quiera escribir en el. No obstante, el *kernel* puede escribir en cualquier posición de el área *libre* desde cualquier zombi sin tener que mapear esta área.

Además mapean una página para datos y código en nivel 3 con permisos de lectura/escritura, junto con otras 8 páginas mas con los mismos permisos. Estas ultimas corresponden a todas las paginas que se encuentran alrededor de la página mapeada en primer lugar (la que contiene el código de la tarea). En la figura 6 se puede ver la posición de las páginas y donde deberán estar mapeadas como direcciones virtuales dentro del área de memoria del zombi. El orden en que estas son mapeadas según la tarea se puede ver en la figura 5.

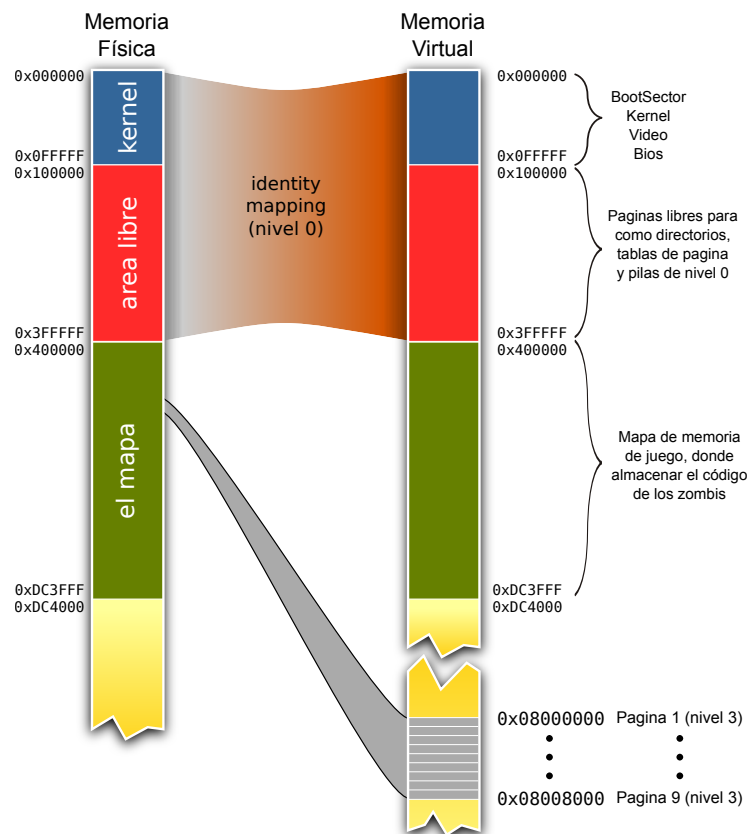


Figura 6: Mapa de memoria de la tarea

La memoria libre será administrada de forma muy simple. Se tendrá un contador de paginas libres a partir de las que se solicita una nueva. Siempre se aumenta en cantidad de páginas usadas y nunca se liberan las páginas pedidas.

3.2. Scheduler

El sistema va a correr tareas de forma concurrente; una a una van a ser asignadas al procesador durante un tiempo fijo denominado *quantum*. El *quantum* será para este scheduler de un *tick* de reloj. Para esto se va a contar con un *scheduler* minimal que se va a encargar de desalojar una tarea del procesador para intercambiarla por otra en intervalos regulares de tiempo.

Como el sistema tiene dos jugadores, los zombis de cada uno de los dos jugadores serán anotados en dos conjuntos distintos. El *scheduler* se encargará de repartir el tiempo entre los dos conjuntos sin importar la cantidad de zombis que tenga cada uno. Esto quiere decir que por cada *tick* de reloj se ejecutará un zombi de cada jugador por vez.

Dado que las tareas pueden generar cualquier tipo de problema, se debe contar con un mecanismo que permita desalojarlas para que no puedan correr nunca más. Este mecanismo debe poder ser utilizado en cualquier contexto (durante una excepción, un acceso inválido a memoria, un error de protección general, etc.) o durante una interrupción porque se llamó de forma incorrecta a un servicio.

Cualquier acción que realice una tarea de forma incorrecta será penada con el desalojo de dicha tarea del sistema, es decir la “muerte” del zombi.

Un punto fundamental en el diseño del *scheduler* es que debe proveer una funcionalidad para intercambiar cualquier tarea por la tarea *Idle*. Este mecanismo será utilizado al momento de llamar al servicio del sistema, ya que la tarea *Idle* será la encargada de completar el *quantum* de la tarea que llamó al servicio. La tarea *Idle* se ejecutará por el resto del *quantum* de la tarea desalojada, hasta que nuevamente se realice un intercambio de tareas por la próxima en la lista.

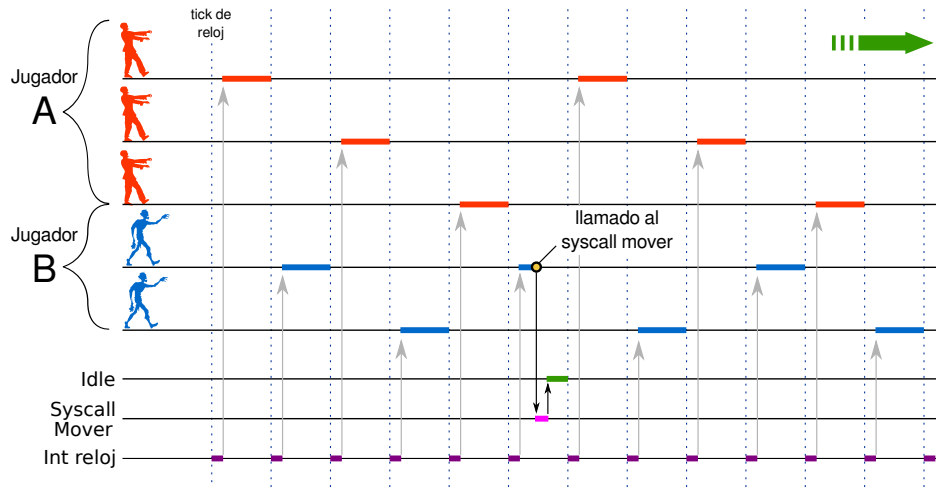


Figura 7: Ejemplo de funcionamiento del *Scheduler*

Inicialmente, la primera tarea en correr es la *Idle*. Luego, en algún momento alguno de los jugadores lanzará algún zombi, lo que implicará que en el próximo *tick* de reloj se comience la ejecución de esta tarea. La misma correrá hasta que termine su tiempo en el próximo *tick* de reloj o la tarea intente llamar al servicio del sistema (moverse); de ser así, será desalojada y el tiempo restante será asignado a la tarea *Idle*. En la figura 7 esto sucede con el primer zombi del jugador B.

3.3. Estructuras para la administración del sistema

El sistema tendrá que almacenar estructuras de datos necesarias para salvar información de las tareas y del juego. Para esto se utilizará una copia del contexto de cada tarea correspondiente a una *tss*.

Además el sistema tendrá control del juego, para esto se deberán salvar algunos datos:

- Posición del jugador y cuál es el tipo de zombi a lanzar
- Posición de cada zombi dentro del mapa
- Zombi/tarea que está siendo actualmente ejecutada y una forma de acceder a la siguiente tarea por ser ejecutada
- Tareas en ejecución y slots libres donde correr nuevas tareas

- Páginas mapeadas por cada tarea

Es decisión de implementación cómo almacenar la información antes listada.

Sin embargo, la memoria en este sistema será administrada de forma muy simple como fue explicado anteriormente.

3.4. Modo debug

El sistema deberá responder a una tecla especial en el teclado, la cual activará y desactivará el modo debugging. La tecla para tal proposito es la “y”. En este modo se deberá mostrar en pantalla la primera excepción capturada por el procesador junto con un detalle de todo el estado del procesador como muestra la figura 8. Una vez impresa en pantalla esta excepción, el juego se detendrá hasta presionar nuevamente la tecla “y” que mantendrá el modo de debug pero borrará la información presentada en pantalla por la excepción. La forma de detener el juego será instantaneamente, al retomar el juego se esperará hasta el próximo ciclo de reloj en el que se decida cuál es la proxima tarea a ser ejecutada. Se recomienda hacer una copia de la pantalla antes de mostrar el cartel con la información de la tarea.

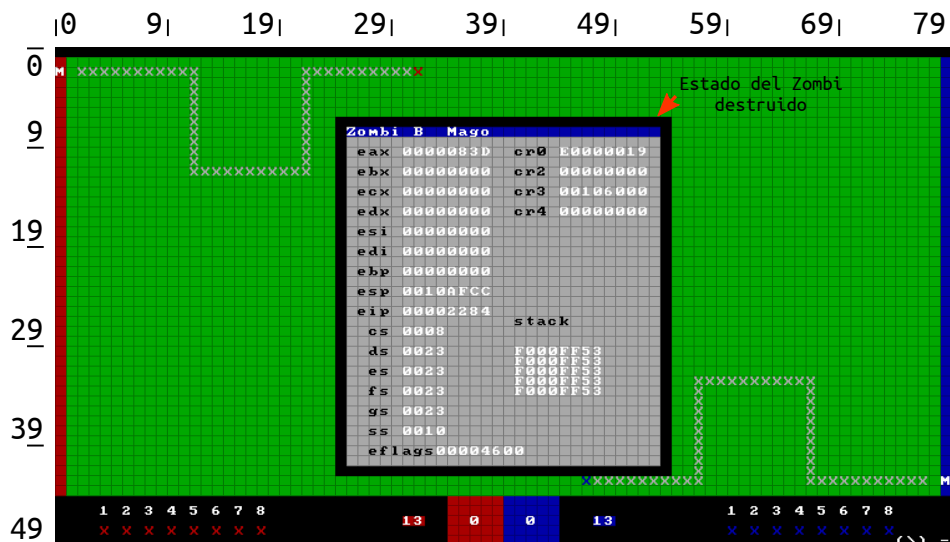


Figura 8: Pantalla de ejemplo de error

3.5. Pantalla

La pantalla presentará un mapa donde se producirá la acción e información del estado de cada jugador y zombis en el sistema.

La figura 9 muestra una imagen ejemplo de pantalla indicando qué datos deben presentarse de forma mínima. Se recomienda implementar funciones auxiliares que permitan imprimir datos en pantalla de forma cómoda. No es necesario respetar la forma de presentar los datos en pantalla, se puede modificar la forma, no así los datos en cuestión.

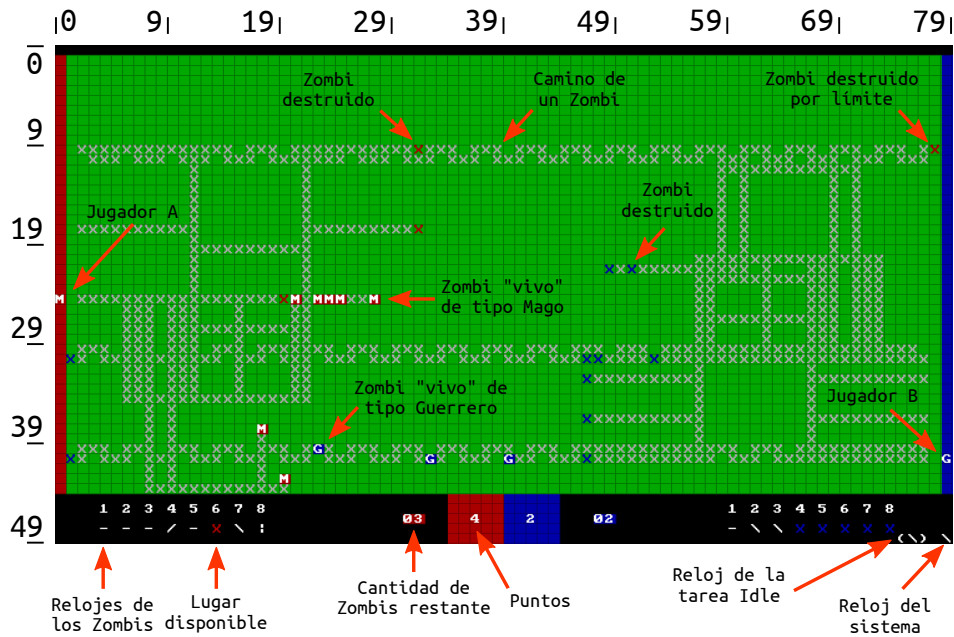


Figura 9: Pantalla de ejemplo

4. Ejercicios

4.1. Ejercicio 1

- Completar la Tabla de Descriptores Globales (GDT) con 4 segmentos, dos para código de nivel 0 y 3; y otros dos para datos de nivel 0 y 3. Estos segmentos deben direccionar los primeros 623MB de memoria. En la *gdt*, por restricción del trabajo práctico, las primeras 7 posiciones se consideran utilizadas y no deben utilizarse. El primer índice que deben usar para declarar los segmentos, es el 8 (contando desde cero).
- Completar el código necesario para pasar a modo protegido y setear la pila del *kernel* en la dirección 0x27000.
- Declarar un segmento adicional que describa el área de la pantalla en memoria que pueda ser utilizado sólo por el *kernel*.
- Escribir una rutina que se encargue de limpiar la pantalla y pintar el área de *el_mapa* un fondo de color (sugerido verde), junto con las dos barras laterales para cada uno de los jugadores (sugerido rojo y azul). ⁽¹⁾ Para este ejercicio se debe escribir en la pantalla usando el segmento declarado en el punto anterior. Es muy importante tener en cuenta que para los próximos ejercicios se accederá a la memoria de vídeo por medio del segmento de datos de 623MB.

Nota: La GDT es un arreglo de `gdt_entry` declarado sólo una vez como `gdt`. El descriptor de la GDT en el código se llama `GDT_DESC`.

¹http://wiki.osdev.org/Text_UI

4.2. Ejercicio 2

- a) Completar las entradas necesarias en la IDT para asociar diferentes rutinas a todas las excepciones del procesador. Cada rutina de excepción debe indicar en pantalla qué problema se produjo e interrumpir la ejecución. Posteriormente se modificarán estas rutinas para que se continúe la ejecución, resolviendo el problema y desalojando a la tarea que lo produjo.
- b) Hacer lo necesario para que el procesador utilice la IDT creada anteriormente. Generar una excepción para probarla.

Nota: La IDT es un arreglo de `idt_entry` declarado solo una vez como `idt`. El descriptor de la IDT en el código se llama `IDT_DESC`. Para inicializar la IDT se debe invocar la función `idt_inicializar`.

4.3. Ejercicio 3

- a) Escribir una rutina que se encargue de limpiar el *buffer* de vídeo y pintarlo como indica la figura 9. Tener en cuenta que deben ser escritos de forma genérica para posteriormente ser completados con información del sistema. Además considerar estas imágenes como sugerencias, ya que pueden ser modificadas a gusto según cada grupo mostrando siempre la misma información.
- b) Escribir las rutinas encargadas de inicializar el directorio y tablas de páginas para el *kernel* (`mmu_inicializar_dir_kernel`). Se debe generar un directorio de páginas que mapee, usando *identity mapping*, las direcciones `0x00000000` a `0x003FFFFFFF`, como ilustra la figura 6. Además, esta función debe inicializar el directorio de páginas en la dirección `0x27000` y las tablas de páginas según muestra la figura 1.
- c) Completar el código necesario para activar paginación.
- d) Escribir una rutina que imprima el nombre del grupo en pantalla. Debe estar ubicado en la primer línea de la pantalla alineado a derecha.

4.4. Ejercicio 4

- a) Escribir una rutina (`inicializar_mmu`) que se encargue de inicializar las estructuras necesarias para administrar la memoria en el área libre (un contador de páginas libres).
- b) Escribir una rutina (`mmu_inicializar_dir_zombi`) encargada de inicializar un directorio de páginas y tablas de páginas para una tarea, respetando la figura 6. La rutina debe copiar el código de la tarea a su área asignada, es decir la posición indicada por el jugador dentro de *el_mapa* y mapear dichas páginas a partir de la dirección virtual `0x08000000` (128MB). Recordar que los zombis comienzan en la segunda columna de *el_mapa* y en la fila correspondiente a la posición donde está el jugador. Sugerencia: agregar a esta función todos los parámetros que considere necesarios.
- c) Escribir dos rutinas encargadas de mapear y desmapear páginas de memoria.

I- `mmu_mapear_pagina(unsigned int virtual, unsigned int cr3, unsigned int fisica)`
Permite mapear la página física correspondiente a `fisica` en la dirección virtual `virtual` utilizando `cr3`.

II- `mmu_unmapear_pagina(unsigned int virtual, unsigned int cr3)`
Borra el mapeo creado en la dirección virtual `virtual` utilizando `cr3`.

- d) Construir un mapa de memoria para tareas e intercambiarlo con el del *kernel*, luego cambiar el color del fondo del primer caracter de la pantalla y volver a la normalidad. Este item no debe estar implementado en la solución final.

Nota: Por la construcción del *kernel*, las direcciones de los mapas de memoria (`page directory` y `page table`) están mapeadas con *identity mapping*. En los ejercicios en donde se modifica el directorio o tabla de páginas, hay que llamar a la función `tlbflush` para que se invalide la *cache* de traducción de direcciones.

4.5. Ejercicio 5

- a) Completar las entradas necesarias en la IDT para asociar una rutina a la interrupción del reloj, otra a la interrupción de teclado y por último una a la interrupción de software `0x66`.
- b) Escribir la rutina asociada a la interrupción del reloj, para que por cada *tick* llame a la función `screen_proximo_reloj`. La misma se encarga de mostrar cada vez que se llame, la animación de un cursor rotando en la esquina inferior derecha de la pantalla. La función `proximo_reloj` está definida en `isr.asm`.
- c) Escribir la rutina asociada a la interrupción de teclado de forma que si se presiona cualquiera de las teclas a utilizar en el juego, se presente la misma en la esquina superior derecha de la pantalla.
- d) Escribir la rutina asociada a la interrupción `0x66` para que modifique el valor de `eax` por `0x42`. Posteriormente este comportamiento va a ser modificado para atender el servicio del sistema.

4.6. Ejercicio 6

- a) Definir las entradas en la GDT que considere necesarias para ser usadas como descriptores de TSS. Minimamente, una para ser utilizada por la `tarea_inicial` y otra para la tarea `Idle`.
- b) Completar la entrada de la TSS de la tarea `Idle` con la información de la tarea `Idle`. Esta información se encuentra en el archivo `TSS.C`. La tarea `Idle` se encuentra en la dirección `0x00016000`. La pila se alojará en la misma dirección que la pila del *kernel* y será mapeada con *identity mapping*. Esta tarea ocupa 1 pagina de 4KB y debe ser “mapeada” con *identity mapping*. Además la misma debe compartir el mismo `CR3` que el *kernel*.
- c) Construir una función que complete una TSS libre con los datos correspondientes a una tarea (zombi). El código de las tareas se encuentra a partir de la dirección `0x00010000` ocupando una pagina de 4kb cada una según indica la figura 1. Para la dirección de la pila se debe utilizar el mismo espacio de la tarea, la misma crecerá desde la base de

la tarea. Para el mapa de memoria se debe construir uno nuevo utilizando la función `mmu_inicializar_dir_zombi`. Además, tener en cuenta que cada tarea utilizará una pila distinta de nivel 0, para esto se debe pedir una nueva pagina libre a tal fin.

- d) Completar la entrada de la GDT correspondiente a la `tarea_inicial`.
- e) Completar la entrada de la GDT correspondiente a la tarea `Idle`.
- f) Escribir el código necesario para ejecutar la tarea `Idle`, es decir, saltar intercambiando las TSS, entre la `tarea_inicial` y la tarea `Idle`.

Nota: En `tss.c` están definidas las `tss` como estructuras TSS. Trabajar en `tss.c` y `kernel.asm` .

4.7. Ejercicio 7

- a) Construir una función para inicializar las estructuras de datos del *scheduler*.
- b) Crear la función `sched_proximo_indice()` que devuelve el índice en la GDT de la próxima tarea a ser ejecutada. Construir la rutina de forma devuelva una tarea de cada jugador por vez según se explica en la sección 3.2
- c) Modificar la rutina de la interrupción `0x66`, para que implemente el servicio *mover* según se indica en la sección 3.1.1.
- d) Modificar el código necesario para que se realice el intercambio de tareas por cada ciclo de reloj. El intercambio se realizará según indique la función `sched_proximo_indice()`.
- e) Modificar las rutinas de excepciones del procesador para que desalojen a la tarea que estaba corriendo y corran la próxima.
- f) Implementar el mecanismo de debugging explicado en la sección 3.4 que indicará en pantalla la razón del desalojo de una tarea.

Nota: Se recomienda construir funciones en C que ayuden a resolver problemas como convertir direcciones de *el_mapa* a direcciones físicas o buscar el proximo zombi a ejecutar.

4.8. Ejercicio 8 (optativo)

- a) Crear un conjunto de 3 ~~tareas~~ zombis (Guerrero, Mago y Clerigo). Los mismos deberán respetar las restricciones del trabajo practico, ya que de no hacerlo no podrán ser ejecutados en el sistema implementado por la cátedra.

Deben cumplir:

- No ocupar más de 4 kb cada uno (tener en cuenta la pila).
- Tener como punto de entrada la dirección cero.
- Estar compilado para correr desde la dirección `0x08000000`.
- Utilizar el unico servicio del sistema (*mover*).

Explicar en pocas palabras qué estrategia utiliza cada uno de los zombis, o en su conjunto en términos de “defensa” y “ataque”.

- b) Si consideran que sus tareas pueden hacer algo mas que completar el primer item de este ejercicio, y tienen a un audaz campion que se atreva a enfrentarse en el campo de batalla zombi, entonces pueden enviar el **binario** de sus tareas a la lista de docentes indicando los siguientes datos,

- Nombre del campion (Alumno de la materia que se presente como “jugador”)
- Nombre de cada uno de las tareas zombi
- Estrategia de alimentación de los zombis (es decir, como se comerán los cerebros de las otras tareas)

Se realizará una competencia a fin de cuatrimestre con premios en/de chocolate para los primeros puestos.

- c) Pelicula y Video Juego favorito sobre Zombis.

5. Entrega

Este trabajo práctico esta diseñado para ser resuelto de forma gradual.

Dentro del archivo **kernel.asm** se encuentran comentarios (que muestran las funcionalidades que deben implementarse) para resolver cada ejercicio. También deberán completar el resto de los archivos según corresponda.

A diferencia de los trabajos prácticos anteriores, en este trabajo está permitido modificar cualquiera de los archivos proporcionados por la cátedra, o incluso tomar libertades a la hora de implementar la solución; siempre que se resuelva el ejercicio y cumpla con el enunciado. Parte de código con el que trabajen está programado en ASM y parte en C, decidir qué se utilizará para desarrollar la solución, es parte del trabajo.

Se deberá entregar un informe que describa **detalladamente** la implementación de cada uno de los fragmentos de código que fueron construidos para completar el kernel. En el caso que se requiera código adicional también debe estar descripto en el informe. Cualquier cambio en el código que proporcionamos también deberá estar documentado. Se deberán utilizar tanto pseudocódigos como esquemas gráficos, o cualquier otro recurso pertinente que permita explicar la resolución. Además se deberá entregar en soporte digital el código e informe; incluyendo todos los archivos que hagan falta para compilar y correr el trabajo en Bochs.

La fecha de entrega de este trabajo es **23/06** y deberá ser entregado a través de un repositorio GIT almacenado en <https://git.exactas.uba.ar> respetando el protocolo enviado por mail y publicado en la página web de la materia.

Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes orga2-doc@dc.uba.ar.