Trabajo Práctico 3
System Programming - `Saltadores y Cazadores`

Organización del Computador 2

Primer Cuatrimestre 2018

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 10 tareas concurrentemente a nivel de usuario. Las tareas estarán distinguidas como saltadoras y cazadoras. Las tareas cazadoras, podrán escribir o leer código de las tareas saltadoras con el objetivo de eliminarlas o capturarlas.

Además el sistema será capaz de capturar cualquier problema que puedan generar las tareas y tomar las acciones necesarias para quitarla o reiniciarla.

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.

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.

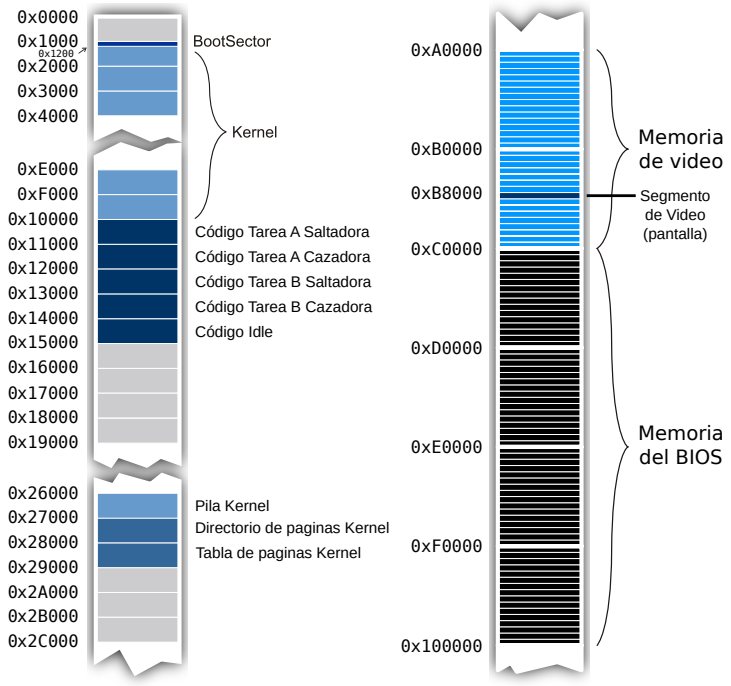

Figura 1: Mapa de la organización de la memoria física del *kernel*

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 `resetear_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.

3. Saltadores y Cazadores

Este trabajo práctico consiste en un juego entre tareas denominadas *Saltadoras* y *Cazadoras*. Cada jugador cuenta con una tarea Saltadora y cuatro tareas Casadoras. Las tareas Casadoras poseen dos servicios, uno para leer y otro para escribir en la memoria de una tarea Saltadora.

Las tareas *Saltadoras* por su parte, deben hacer todo lo posible para no ser capturadas. Para esto deben saltar por toda la memoria disponible esperando que su PC no sea capturado por el código escrito por las tareas Cazadoras.

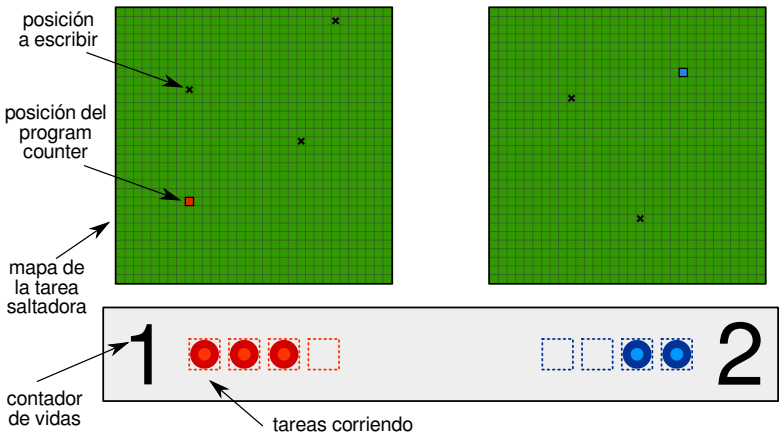

Figura 2: El Juego

3.1. Tareas Cazadoras y Saltadoras

Las tareas *Cazadoras* dispondrán de tres servicios del sistema descriptos a continuación:

syscall	parámetros	descripción

`escribir`	`in EAX=0x542`
	`in EBX=uint32_t offset`
	`in ECX=uint32_t* dato`
		Escribe cuatro bytes desde `dato` en dirección de la tarea Salvadora contrincante indicada por `offset`.

`leer`	`in EAX=0x7CA`
	`in EBX=uint32_t offset`
	`in ECX=uint32_t* dato`
		Lee cuatro bytes desde la tarea Salvadora contrincante en la dirección indicada por `offset` y guarda la lectura en `dato`.

`numero`	`in EAX=0x824` `out EBX=uint32_t numero`	Escribe en el registro `EBX` el número de tarea Cazadora (`1`, `2`, `3` o `4`).

Estos servicios podrán ser utilizandos solamente desde una tarea Cazadora, en el caso que una tarea Saltadora intente utilizar uno de los servicios, debe ser desalojada.

Considerar que tanto el servicio de `leer` como `escribir` pueden ser llamados solamente con direcciones y desplazamientos válidos. Teniendo en cuenta que la unidad direccionable en ambos casos es de 4 bytes, el puntero a dato debe estar en el rango de direccionamiento de una pagina (0x08000000 a 0x08000FFC) y el deplazamiento no puede superar el tamaño en bytes de una pagina (0 a 4092).

Además el comportamiento de los servicios una vez ejecutado es diferente dependiendo del servicio. Para el caso de `leer` y `numero`, una vez ejecutado, se retorna a la ejecución de la tarea que lo solicito. En cambio, para el servicio `escribir`, se debe comenzar a ejecutar la tarea `Idle` durante el resto del *quantum*. Este último comportamiento es ilustrado por la figura 4.

3.1.1. Lógica de juego

Inicialmente todas las tareas Cazadoras y Saltadoras comienzan a ser ejecutadas respetando lo explicado en la sección 3.2. Si una tarea Cazadora produce un error, es desalojada y nunca más vuelve a correr. Por otro lado si una tarea Saltadora produce un error, es desalojada y vuelta a cargar inmediatamente. Las tareas Saltadoras pueden ser cargadas una cantidad limitada de veces indicada por el máximo de vidas disponibles, inicialmente configurado en 5.

El juego termina en dos casos. El primero, es cuando la cantidad de vidas de una tarea Saltadora se terminen, en este caso gana el jugador tal que su tarea Saltadora aun tiene al menos una vida. El segundo caso es cuando todas las tareas Cazadoras de uno de los jugadores fueron eliminadas. En este último caso, el ganador es el jugador que aun tiene tareas Cazadoras ejecutando.

3.1.2. Organización de la memoria

Cada una de las tareas, ya sean Saltadoras o Cazadoras, tiene mapeadas las áreas de *kernel* y *libre de kernel* con *identity mapping* en nivel 0. Sin embargo, área *libre tareas* no está mapeada. Esto obliga al *kernel* a mapear el dicha área cada vez que quiera escribir en la misma. No obstante, el *kernel* puede escribir en cualquier posición del área *libre de kernel* desde cualquier tarea sin tener que mapearla. Además, para código y datos, mapean una página en nivel 3 con permisos de lectura/escritura según indica la figura 3.

La memoria libre de kernel y memoria de libre de tareas será administrada de forma muy simple. Se tendrá un contador de paginas por cada una de estas áreas, a partir del cual se solicitará una nueva página. Este contador se aumentará siempre 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 y dos tipos de tareas, cada uno de los dos jugadores serán anotados en dos conjuntos distintos. El *scheduler* se encargará de repartir el tiempo

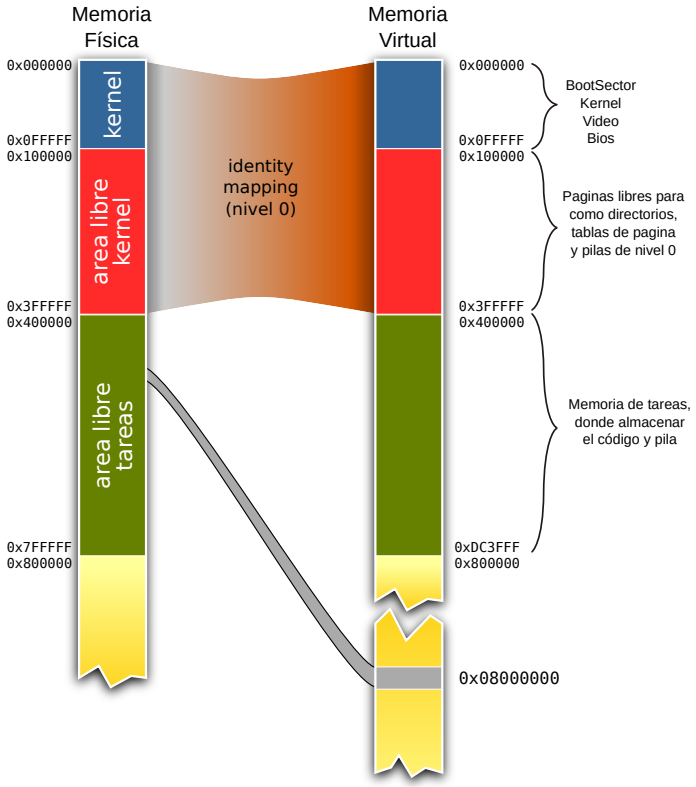

Figura 3: Mapa de memoria de la tarea

entre los dos conjuntos sin importar la cantidad de tareas que tenga cada uno. Esto quiere decir que por cada *tick* de reloj se ejecutará una tarea de cada jugador por vez.

Además, las tareas saltadoras serán ejecutadas cada dos turnos, mientras que las cazadoras serán ejecutadas una por vez en el turno restante. Este comportamiento es ilustrado en la figura 4.

Ya sean saltadoras o cazadoras, las tareas pueden generar cualquier tipo de problema. Se debe entonces 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.

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.

Inicialmente, la primera tarea en correr es la `Idle`. Luego, en el primer intercambio se comenzará a correr una de las tareas. La misma correrá hasta que termine su tiempo en el próximo *tick* de reloj o la tarea intente llamar a uno de los servicios del sistema; de ser así,

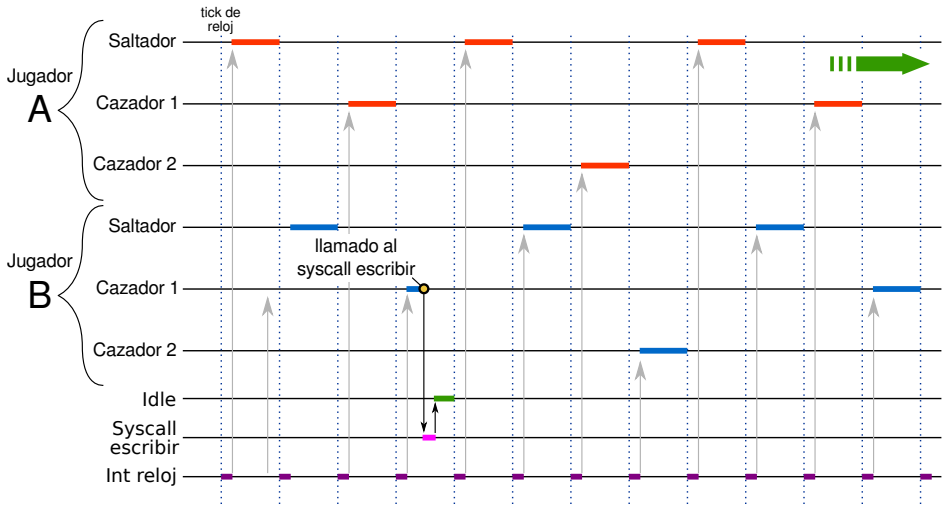

Figura 4: Ejemplo de funcionamiento del *Scheduler*, considerando solo dos tareas Cazadoras por jugador.

será desalojada y el tiempo restante será asignado a la tarea `Idle`.

3.3. 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 5. 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 próxima tarea a ser ejecutada. Se recomienda hacer una copia de la pantalla antes de mostrar el cartel con la información de la tarea.

3.4. Pantalla

La pantalla presentará dos mapas, uno para cada tarea Saltadora. El mapa de 32×32 representará toda la memoria de la tarea, donde cada caracter corresponde a 4 bytes de memoria. En este mapa se indicará en color la posición del `PC`, y con el caracter `x` la posición donde una tarea Cazadora escribió. Ambas indicaciones serán expuestas en cada ciclo de reloj.

La figura 6 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.

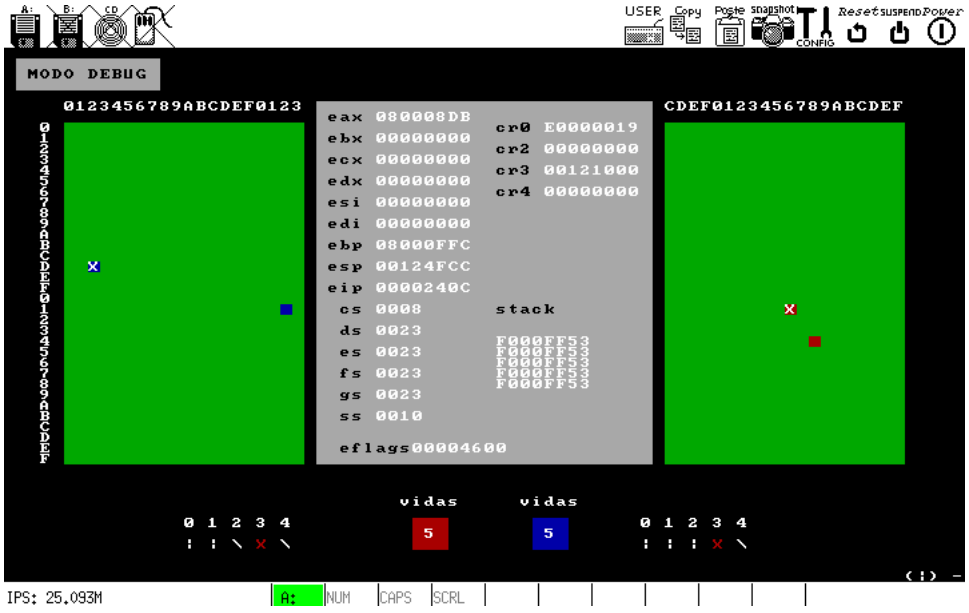

Figura 5: Pantalla de ejemplo de error

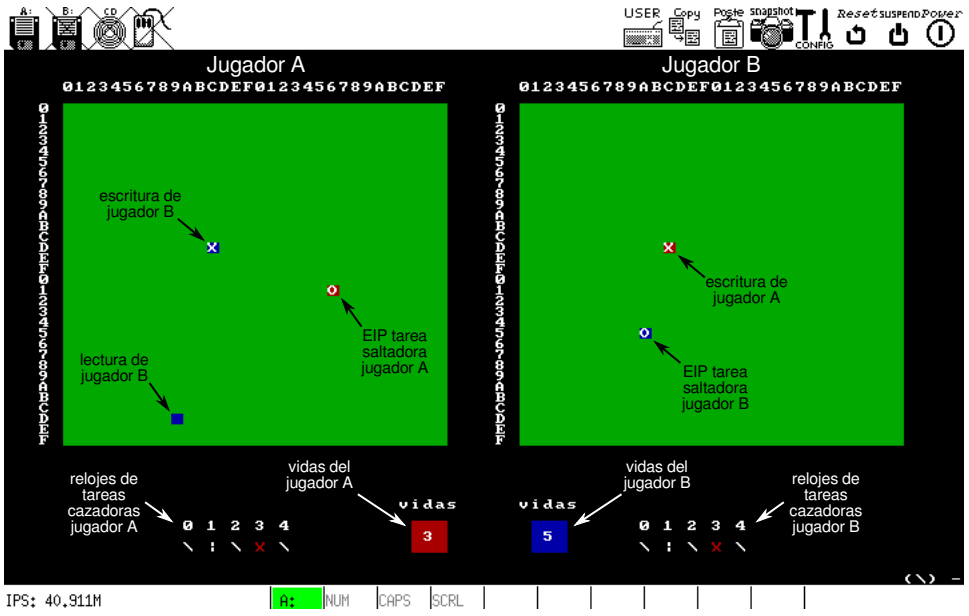

Figura 6: Pantalla de ejemplo

4. Ejercicios

4.1. Ejercicio 1

a) 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 314MB de memoria. En la *gdt*, por restricción del trabajo práctico, las primeras 19 posiciones se consideran utilizadas y no deben utilizarse. El primer índice que deben usar para declarar los segmentos, es el 20 (contando desde cero).

b) Completar el código necesario para pasar a modo protegido y setear la pila del *kernel* en la dirección `0x27000`.

c) Declarar un segmento adicional que describa el área de la pantalla en memoria que pueda ser utilizado sólo por el *kernel*.

d) Escribir una rutina que se encargue de limpiar la pantalla y pintar en pantalla[1] las áreas de los mapas con algún color de fondo, junto con las barras de los jugadores según indica la sección 3.4. 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 314MB.

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`.

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) 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 `proximoReloj`. 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 `proximoReloj` está definida en `isr.asm`.

c) Escribir la rutina asociada a la interrupción de teclado de forma que si se presiona cualquiera de `0` a `9`, 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.

[1] http://wiki.osdev.org/Text_UI

4.4. Ejercicio 4

a) 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 `0x003FFFFF`, como ilustra la figura 3. Además, esta función debe inicializar el directorio de páginas en la dirección `0x27000` y las tablas de páginas según muestra la figura 1.

b) Completar el código necesario para activar paginación.

c) Escribir una rutina que imprima el número de libreta de todos los integrantes del grupo en el extremo superior izquierdo de la pantalla.

4.5. Ejercicio 5

a) Escribir una rutina (`mmu_inicializar`) que se encargue de inicializar las estructuras necesarias para administrar la memoria en el área libre de kernel y en el área libre de tareas (dos contadores de paginas libres).

b) Escribir una rutina (`mmu_inicializarDirTarea`) encargada de inicializar un directorio de páginas y tablas de páginas para una tarea, respetando la figura 3. La rutina debe solicitar una página para tareas libre donde copiar el código de la tarea y mapear dicha página a partir de la dirección virtual `0x08000000`(128MB). Sugerencia: agregar a esta función todos los parámentros que considere necesarios.

c) Escribir dos rutinas encargadas de mapear y desmapear páginas de memoria.

I- `mmu_mapearPagina(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_unmapearPagina(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 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, se debe que llamar a la función `tlbflush` para que se invalide la *cache* de traducción de direcciones.

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

`0x00014000`. 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, ya sea Saltadora o Cazadora. 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_inicializarDirTarea`. Además, tener en cuenta que cada tarea utilizará una pila distinta de nivel 0, para esto se debe pedir una nueva pagina del área libre de kernel 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_proximoIndice()` 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 los servicios del sistema según se indica en la sección 3.1.

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_proximoIndice()`.

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 3.3 que indicará en pantalla la razón del desalojo de una tarea.

4.8. Ejercicio 8 (optativo)

a) Crear una tarea Saltadora y una tarea Cazadora. Las mismas 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 una (tener en cuenta la pila).

- Tener como punto de entrada la dirección cero.
- Estar compilado para correr desde la dirección `0x08000000`.
- Utilizar los servicios del sistema correctamente.

Explicar en pocas palabras qué estrategia utiliza cada una de las tareas como "defensa" y "ataque" según corresponda.

b) Si consideran que sus tareas pueden hacer algo más que completar el primer ítem de este ejercicio, y se atreven a enfrentarse a una cacería de PC, entonces pueden enviar el **binario** de sus tareas a la lista de docentes indicando los siguientes datos el nombre de la tarea.

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

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 **19/06**. 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.