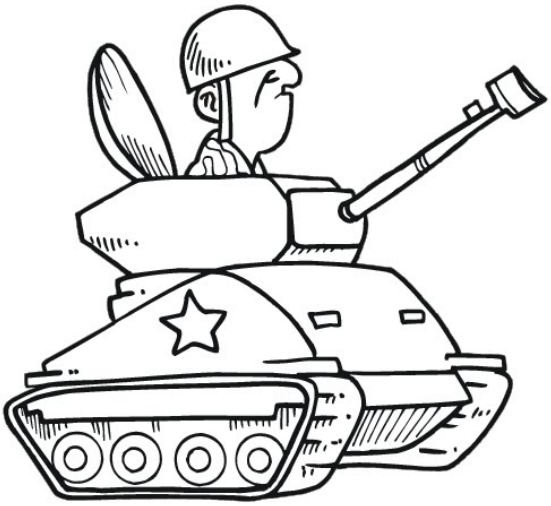
Trabajo Práctico 3
System Programming - `TronTank`

Organización del Computador 2

Primer Cuatrimestre 2014



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 exactamente 8 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.

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.
- `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.

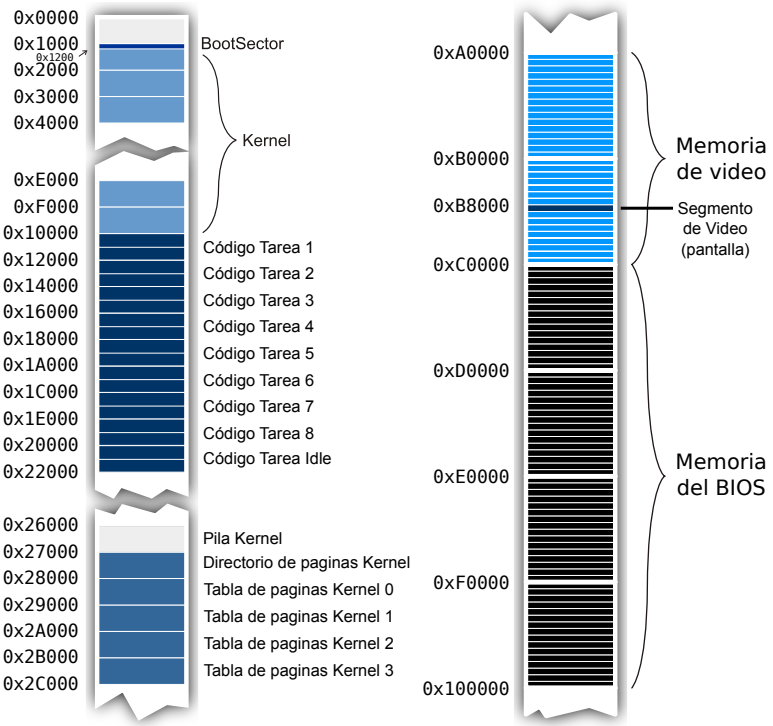


Figura 1: Mapa de la organización de la memoria física del *kernel*

3. TankFight

La megacorporación ENCOM ha reforzado la seguridad sobre el *Master Control Program* a pedido del ejecutivo Dillinger. Éste, cansado de las andanzas de Kevin Flynn, ha resuelto liberar tanques de luz en el sistema raiz. Si bien Flynn sabe andar muy bien en motos de luz, no tiene ni la mas remota idea como se manejan tanques de luz.

En este trabajo práctico vamos a implementar un sistema de práctica para tanques de luz, obviamente controlados por la inteligencia artificial mas sofisticada.

Los tanques o tareas, se moverán sobre un espacio infinito mapeando páginas de memoria a su paso. Van a tener dos herramientas de defensa, poner minas o disparar misiles. En la figura 2 se puede dar cuenta graficamente de las posibles acciones de las tareas-tanque.

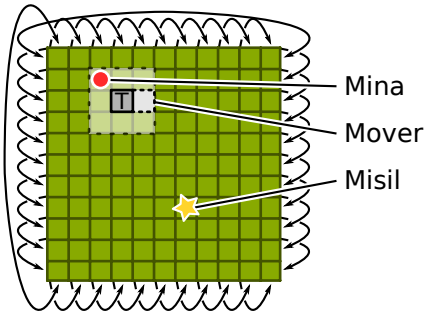


Figura 2: Posibilidades de acciones para un tanque

La memoria del sistema estará dividida en tres secciones, la primera, *kernel* contendrá todas las estructuras y código de administración del sistema. La segunda, el *area_libre* será el espacio que se utilice para solicitar memoria dinamicamente y por último, *el_mapa* será el area donde se moverán los tanques.

3.1. Tanks (Tareas)

El sistema correrá 8 tareas o tanques concurrentemente. Los mismos podrán realizar tres acciones como fue mencionado anteriormente, “desplazamientos por el mapa”, “lanzar misiles” o “colocar minas anti tanques”. Estas acciones serán descriptas más adelante como servicios que provee el sistema para con los tanques.

Cada tanque tendrá asignadas inicialmente dos páginas de memoria de 4kb, estas paginas estarán ubicadas dentro del area de memoria denominada *el_mapa*. Éste corresponderá a un area de memoria física de 50×50 paginas de 4kb.

Inicialmente los tanques solo podrán acceder a sus únicas dos páginas asignadas de memoria física dentro de *el_mapa*.

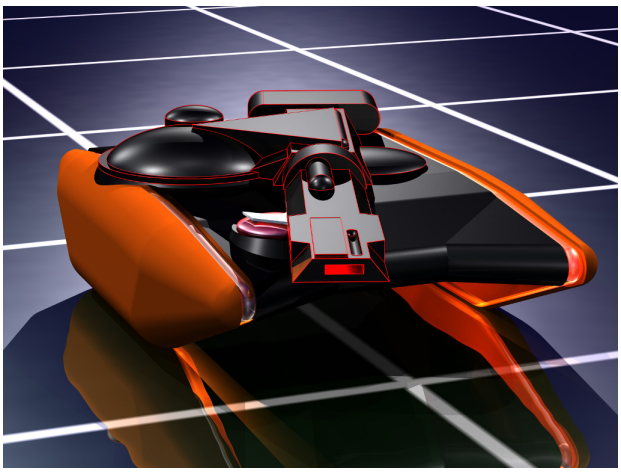


Figura 3: Ejemplo de tank de luz

A medida que los tanques se desplacen por *el_mapa*, cada desplazamiento corresponderá a una pagina de memoria física, la cual será mapeada linealmente dentro del area virtual del tanque en cuestión.

3.1.1. Controles del tanque (Servicios del Sistema)

Los tanques podrán solicitar tres servicios al sistema (acciones o controles del tanque). Los servicios (*system calls*) se solicitarán por medio de la interrupción `0x52`. El registro `EAX` indicará el número del servicio pedido, mientras que los registros `EBX`, `ECX` y `EDX` serán los parámetros del servicio.

A continuación se describe el funcionamiento de cada uno de los servicios. En la figura 4 se ilustra el comportamiento de los tres llamados al sistema.

- **"desplazamientos por el mapa" (mapear una pagina de memoria)**

  El código de cada tanque está mapeado incialmente en dos páginas de memoria a partir de la dirección correspondiente a los 128mb de memoria virtual. A partir de esta dirección, luego de las dos paginas mapeadas inicialmente, se irán mapeando las paginas de memoria física que el tanque recorra. Si a la hora de mapear una nueva pagina física, esta está ya esta mapeada dentro del espacio del tanque, entonces no se mapea nuevamente.

  El servicio se encargará de

  - mostrar en pantalla la nueva posición a la que se mueve el tanque
  - mapear la página correspondiente a la próxima página no mapeada del espacio de la tarea
  - comprobar que no haya ninguna mina en esa página, de existir el tanque será destruido.

  Una vez solicitado este servicio el tanque pierde su turno, esto quiere decir que es removido del scheduler para dar paso a la siguiente tarea.

  → `SYS_MOVER`

  `EAX = 0x83D`
  `EBX =` dirección de desplazamiento **(*)**

- **"lanzar misiles" (escribir en una pagina no mapeada)**

  Los tanques tvenen cañones que les permiten disparar balas a sus enemigos, ahora, como los disparos serán dirigidos a posiciones precisas del mapa, los llamaremos misiles. Dicho esto, este servicio permitirá lanzar misiles a posiciones de *el_mapa* direccionadas de forma relativa a la posición del tanque. Los valores posibles para `x` e `y` no pueden superar en módulo el tamaño maximo de *el_mapa*, es decir 50.

  El *buffer* donde se almacena el misil debe ser un rango dentro del area mapeada por la tarea, además el tamaño del mismo no debe superar los 4096 bytes.

  El servicio se debe encargar de copiar el buffer del misil dentro al principio de la pagina física idenfiticada por las coordenadas `x` e `y` dentro de *el_mapa*. Una vez lanzado el misil, el scheduler dará paso a la próxima tarea.

→ SYS_MISIL

EAX = 0x911
EBX = desplazamiento relativo en x
ECX = desplazamiento relativo en y
EDX = dirección virtual del buffer del misil
ESI = tamaño del misil en bytes (maximo=4096)

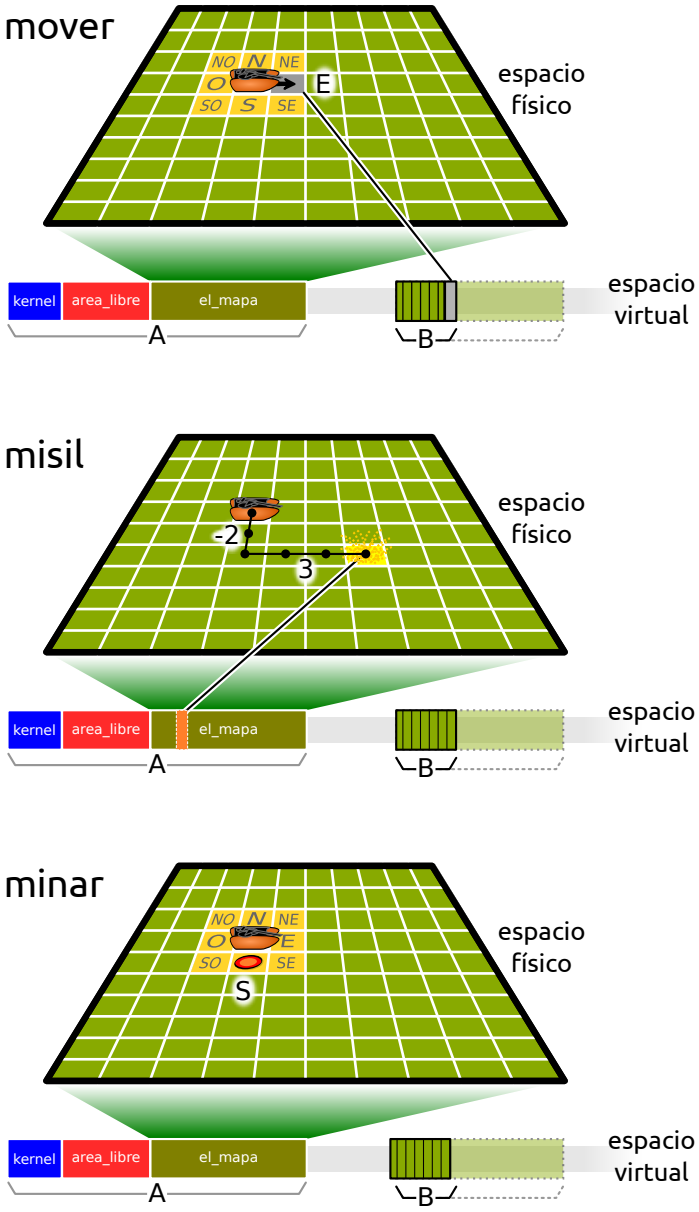


Figura 4: Ejemplos de Syscalls

- **"colocar minas anti tanques" (marcar como memoria mala)**

  Además de lanzar misiles los tanques pueden colocar minas anti-tanques. Estas minas destruyen a los tanques que se quieran mover a la posición donde esta la mina. Las minas duran una sola vez, es decir, si un tanque es destruido por una mina, ésta desaparece (tambien es destruida). Las minas también pueden ser destruidas si un misil cae sobre la misma.

  El único parámetro de este servicio es la posición donde colocar la mina. Un tanque puede colocar una mina en alguna de las 4 posibles direcciones a su alrededor de la posición actual donde se encuentra.

  Una vez llamado este servicio el scheduler desalojara al tanque y ejecutará la próxima tarea mientras se encarga de colocar la mina anti-tanque en su lugar.

  → `SYS_MINAR`

  `EAX = 0x355`
  `EBX =` dirección de colocación (*)

La figura 4 muestra un ejemplo de cada uno de los llamados al sistema. En el ejemplo de `mover` se desplaza el tanque hacia la derecha y se mapea la página indicada en el area de memoria virtual del tanque. En el ejemplo de `misil` se lanza un misil a la posición relativa `-2, 3`. Se escribe en la página correspondiente dentro del espacio de *el_mapa*. En el ejemplo de `minar` se coloca una mina al sur del tanque.

Una vez llamado cualquiera de estos servicios, el *scheduler* se encargará de desalojar al tanque que lo llamó para dar paso a la próxima tarea. Este mecanismo será detallado mas adelante.

(*)

NE=12	N=11	NO=14
E =22	C=0	O =44
SE=32	S=33	SO=34

3.1.2. Organización de la memoria

Cada uno de los tanques-tarea tiene mapeadas las áreas de *kernel*, *libre* y *mapa* con *identity mapping* en nivel 0, esto permite al *kernel* escribir en cualquier posición de memoria útil desde el contexto de cualquier tarea. Además las tareas mapean dos páginas para datos y código en nivel 3 con permisos de lectura/escritura. En la figura 5 se puede ver la posición de las páginas y donde deberán estar mapeadas como direcciones virtuales dentro del area de memoria del tanque.

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.

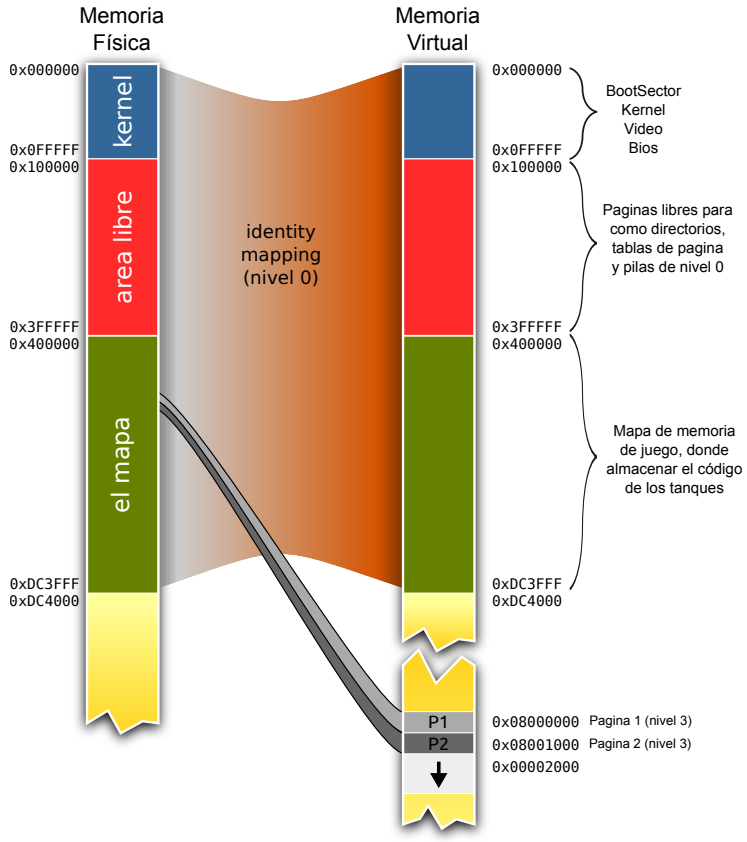


Figura 5: Mapa de memoria de la tarea

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 para acceder a uno de los tres posibles servicios del sistema, porque por ejemplo se intento llamar a un servicio que no existe.

Cualquier acción que realice una tarea de forma incorrecta será penada con el desalojo de dicha tarea del sistema, es decir el destrucción del tanque.

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 a cualquier 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` como se puede ver en la figura 6. Luego, en el primer *tick* de reloj se lanzará la tarea 1. Ésta correrá hasta que termine su tiempo en el próximo *tick* de reloj o la tarea intente llamar a un servicio del sistema; de ser así, será desalojada y el tiempo restante será asignado a la tarea `Idle`. En la figura 6 esto sucede con la tarea 2.

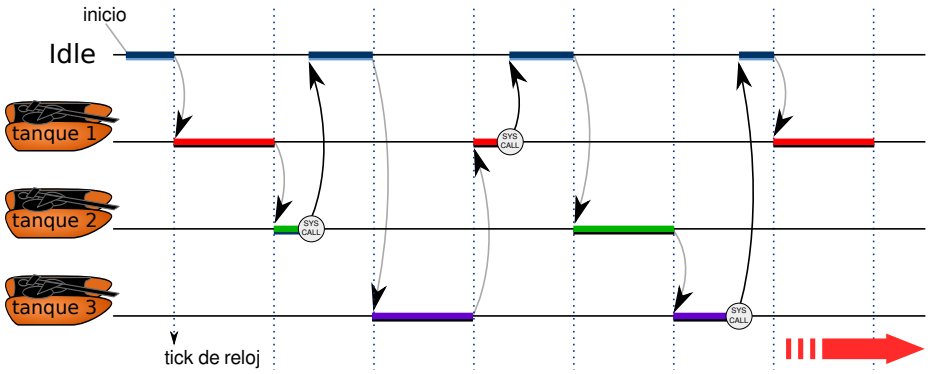


Figura 6: Ejemplo de funcionamiento del *Scheduler*

3.2.1. Mecanismo de Scheduler por swapping

La implementación del *scheduler* consta de tan solo dos estructuras de `tss` dentro de la `gdt`. Una de las estructuras de `tss` la denominaremos `actual` y a la otra `anterior`. En `actual` se almacenará la `tss` que esta corriendo actualmente dentro del sistema, y en `anterior` se almacenará la `tss` que habia sido ejecutada anteriormente. El mecanismo para el intercambio de tareas será el siguiente, cuando llegue una interrupción de reloj se copiará el contenido de `anterior` a una estructura de resguardo para el contenido de las `tss`. Luego se intercambiará `anterior` con `actual` y se almacenará en `actual` el contenido de la `tss` de la próxima tarea a ejecutar.

En la figura 7 se pueden ver un ejemplo del mecanismo de *scheduler*. A continuación se detallan los pasos del ejemplo en relación con el ciclo de ejecución y desalojo de la tarea A.

a - En el primer intercambio se copia el contexto de la tarea A dentro del la tss 2. Esta se ejecuta hasta que se produce una interrupción de reloj.

b - En el próximo intercambio se copia la próxima tarea y se salva el estado de la anterior. La tarea A aun no fue desalojada por completo, ya que la rutina de atención de interrupciones del reloj esta siendo ejecutada en el contexto de la tarea A.

c - Cuando llega la tercer interrupción de reloj, el contexto de la tarea A ya fue guardado dentro de la tss, por lo que puede ser copiado al vector de contextos del sistema.

3.3. Estructuras para la administracion del sistema

En el sistema se tendrá que administrar las estructuras de datos necesarias para guardar información de las tareas y del juego. Para esto se utilizará una copia del contexto de cada tarea que se almacerá en la *tss* correspondiente a la hora de ejecutar dicha tarea. La forma e información que se almacenará en el contexto de cada tarea es decisión de implementación.

Además el sistema tendrá control del juego, para esto se deberán salvar dos conjuntos de datos. Por un lado la posición de las tareas en el mapa y por otro las posiciónes de las minas, y a qué tanque corresponden estas.

Sin embargo, la memoria en este sistema 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.

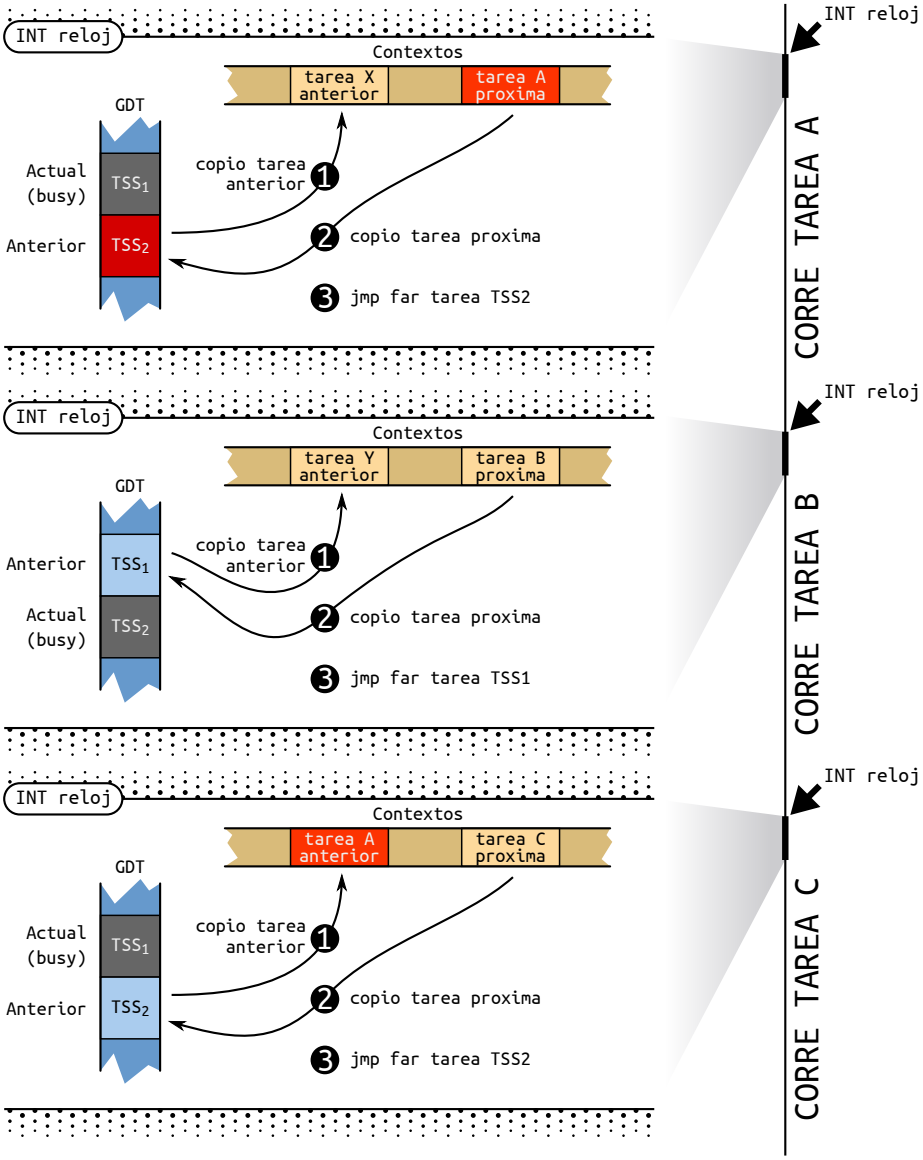


Figura 7: Mecanismo del *scheduler*

3.4. Teclado

La funcionalidad del teclado será muy limitada, nos permitirá solo dos acciones. La primera será la posibilidad de detener el juego mediate el uso de la tecla "p", esta acción pausará el juego. La accion de *pause* **NO** será considerada inmediatamente al presionar la tecla, sino que se detendrá el juego en la primera interrupción de reloj que llegue luego de presionar la tecla. El mismo mecanismo será usado para reactivar el juego, se esperará hasta la próxima interrupción de reloj para determinar que si se puede comenzar a correr otra tarea. El juego en modo *pause* ejecutará la tarea `idle` durante todo el tiempo que ese modo.

La segunda funcionalidad que permitirá el teclado será poder mostrar en pantalla los

errores de las tareas y la razón por la cual fueron desalojadas. Esto se realizará mediante los numeros del “1” al “8”, uno para cada tarea. La información debe ser mostrada según se indica en la sección 3.5.

3.5. Pantalla

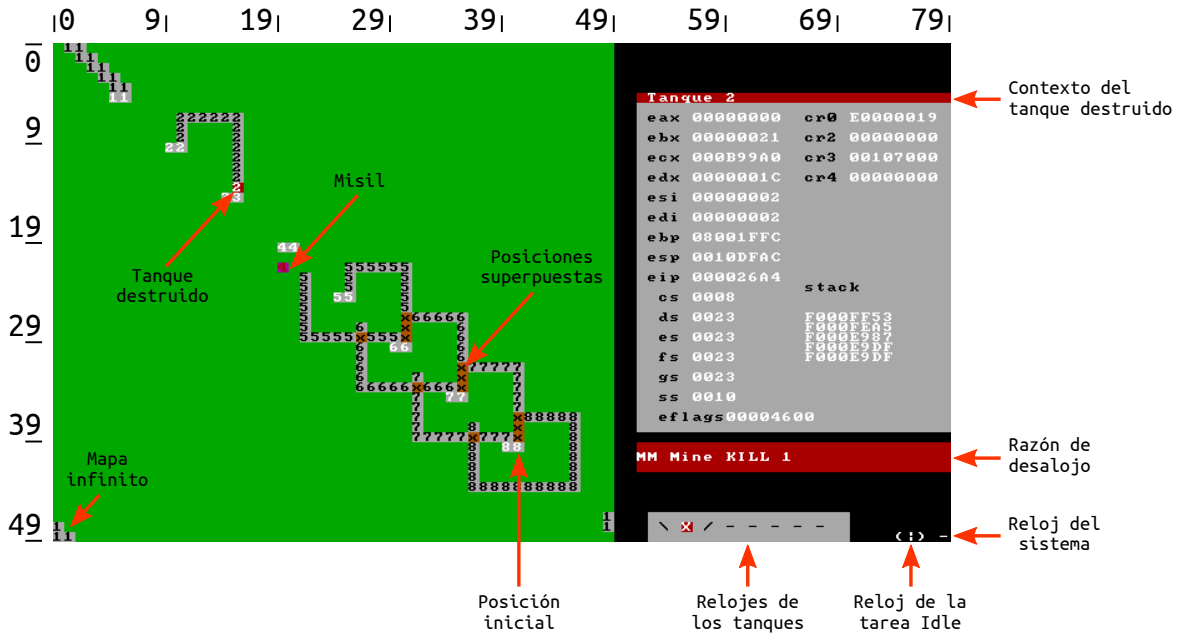


Figura 8: Pantalla de ejemplo

La pantalla presentará un mapa de el area de 50×50 donde se producirá la acción e información del estado de cada tanque en el sistema.

La figura 8 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.

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 733MB de memoria. En la *gdt*, por restricción del trabajo práctico, las primeras

8 posiciones se consideran utilizadas y no deben utilizarse. El primer índice que deben usar para declarar los segmentos, es el 9 (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 el area de *el_mapa* un fondo de color (sugerido verde). Para este ejercicio se debe escribir en la pantalla usando el segmento declarado en el punto anterior (para los próximos ejercicios se accederá a la memoria de vídeo por medio del segmento de datos de 773MB).

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) Escribir una rutina que se encargue de limpiar el *buffer* de vídeo y pintarlo como indica la figura 8. 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 `0x00DC3FFF`, como ilustra la figura 5. 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 linea 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 area libre.

b) Escribir una rutina (`mmu_inicializar_dir_tarea`) encargada de inicializar un directorio de páginas y tablas de páginas para una tarea, respetando la figura 5. La rutina debe copiar el código de la tarea a su área asignada, es decir sus dos páginas de código dentro de *el_mapa* y mapear dichas páginas a partir de la dirección virtual `0x08000000`(128MB).

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.

Nota: Por la construcción del *kernel*, las direcciones de los 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 `0x52`.

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 cualquier número, se presente el mismo en la esquina superior derecha de la pantalla. El número debe ser escrito en color blanco con fondo de color aleatorio por cada tecla que sea presionada [1].

d) Escribir la rutina asociada a la interrupción `0x52` para que modifique el valor de `eax` por $0x42$. Posteriormente este comportamiento va a ser modificado para atender los servicios del sistema.

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

4.6. Ejercicio 6

a) Definir 3 entradas en la `GDT` para ser usadas como descriptores de `TSS`. Una será reservada para la `tarea_inicial` y otras dos para realizar el intercambio entre tareas, denominadas `TSS1` y `TSS2` respectivamente.

b) Completar la entrada de la `TSS1` con la información de la tarea `Idle`. Esta información se encuentra en el archivo `TSS.C`. La tarea `Idle` se encuentra en la dirección `0x00020000`. La pila se alojará en la misma dirección que la pila del kernel y será mapeada con *identity mapping*. Esta tarea ocupa 2 paginas de 4KB y debe ser mapeada con *identity mapping*. Además la misma debe compartir el mismo `CR3` que el *kernel*.

c) Completar el resto de la información correspondiente a cada tarea en la estructura auxiliar de contextos. El código de las tareas se encuentra a partir de la dirección `0x00010000` ocupando dos páginas de 4kb cada una. El mismo debe ser mapeado a partir de la dirección `0x08000000`. 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 para cada tarea utilizando la función `mmu_inicializar_dir_usuario`. 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 `TSS1`, que contiene la información de la tarea `Idle`.

f) Completar la entrada de la `GDT` correspondiente a la `TSS2`.

g) 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 el indice de la `TSS1` y luego el de la `TSS2` de forma intercalada, para dos tareas fijas.

c) Modificar la rutina de la interrupción `0x52`, para que implemente los tres servicios del sistema 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 la función `sched_proximo_indice()` de forma que ejecute todas las tareas según se describe en la sección 3.2.

f) Modificar las rutinas de excepciones del procesador para que impriman el problema que se produjo en pantalla, desalojen a la tarea que estaba corriendo y corran la próxima, indicando en pantalla porque razón fue desalojada la tarea en cuestión.

Nota: Se recomienda construir funciones en C que ayuden a resolver problemas como convertir direcciones de *el_mapa* a direcciones físicas.

4.8. Ejercicio 8 (optativo)

a) Crear un tanque (tarea) propio que mapee paginas a muerte contra otros intrepidos tanques. Para esto pueden editar el código del primer tanque a gusto.

El tanque debe tener las siguientes características,

- No ocupar más de 8 kb (tener en cuenta la pila).
- Tener como punto de entrada la dirección cero.
- Estar compilada para correr desde la dirección `0x08000000`.
- Utilizar solo los servicios presentados en el trabajo práctico.

Explicar en pocas palabras qué estrategia utilizaron en su tanque en términos de "defensa" y "ataque".

b) Si consideran que su tanque es capaz de enfrentarse contra los tanques del resto de sus compañeros, pueden enviar el **binario** a la lista de docentes indicando los siguientes datos,

- Nombre del tanque, *ej: "T-90S Bhishma"*
- Características letales, *ej: Cañón 2A46M de 125 mm con cargador automático*
- Sistema de defensa, *ej: 1.350 mm en blindaje laminado, blindaje reactivo Kontakt-5*

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

c) Mencionar la mayor cantidad de características del tanque montado en el jardín del edificio Libertador en la ciudad de Buenos Aires.

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 **17-06** y deberá ser entregado a través de la página web en un solo archivo comprimido en formato `tar.gz`, con un límite en tamaño de 10Mb. El sistema sólo aceptará entregas de trabajos hasta las **16:59** del día de entrega.

Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes.