DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico III

System Programming

Organización del Computador II

Primer Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Gonzalo Consoli	595/18	gonzaloconsoli@hotmail.com
Julián Recalde Campos	502/17	recaldej@hotmail.com.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En este informe vamos a presentar la implementación de un kernel programado en assembler y en C. En esta implementación además de implementar el código básico para arrancar el sistema operativo, se implementara un juego basado en rick and morty, en el cual hay 3 tipos de tareas: tareas rick, tareas morty, tareas cronenberg, el juego consiste en 2 equipos, rick y morty de dimensión 137 y el otro equipo de la dimensión 248, el objetivo de cada equipo es controlar la mayor cantidad de tareas. se utilizara la herramienta botchs para ejecutar el código y así correr el código que permitirá ver en pantalla este juego.

El código consiste de diferentes secciones que se encargan de cumplir con los diferentes requerimientos que un procesador Intel de 32 bits exige para poder inicializar correctamente un sistema operativo.

2. Modo real, protegido y inicializaciones (kernel.asm)

En modo real (16bits) debemos preparar el sistema para poder pasarlo a modo protegido, para esto deshabilitamos interrupciones, habilitamos el a20,configuramos el modo de vídeo , cargamos la gdt, es importante tener deshabilitadas las interrupciones y cargar al gdt ya que en 32 bits buscara utilizar estos elementos.

Pasamos a modo protegido (32 bits), esto lo logramos haciendo un jump far seleccionando un segmento. En este modo, establecemos todas las estructuras que vamos a utilizar en el resto del código, establecemos:

los selectores de segmento,la base de la pila, inicializamos pantalla, y el manejo de memoria de paginación. para esto inicializamos el directorio de paginas, y activamos paginación.

Inicializamos la IDT para poder activar las interrupciones, y configuramos los controladores de interrupciones.

Inicializamos el tss,y la tarea idle, inicializaremos la tarea inicial para el Scheduler y el Scheduler, que permitirá alternar las distintas tareas utilizando interrupciones de reloj anterior mente programadas y especificadas en la IDT.

Activamos las interrupciones y finalmente comenzamos a ejecutar las tareas que son alternadas por el scheduler.

A continuación explicamos con mas detalle como inicializamos cada estructura y su código respectivo.

3. estructuras y componentes del kernel

3.1. gdt (gdt.c)

En gdt implementamos las entradas de la gdt, tenemos los segmentos de códigos,datos de nivel 0 y 3 que nos piden con los atributos del enunciado, estos segmentos tienen base cero y ocupan toda la memoria ram hasta el limite de 0x88ff .Por otro lado tenemos un descriptor de segmento que se utiliza para mapear la memoria con la pantalla, a través de este segmento editando cada posición de memoria dentro de este uno edita cada píxel mostrado en pantalla, este segmento de pantalla comienza en 0xb8000 y tiene limite de 1f3f como indica en el enunciado, privilegio cero con tipo read/write para poder modificar la pantalla desde el kernel.

Luego tenemos los descriptores de tss de las tareas, la tarea inicial, para tener una primer tarea donde podamos dejar la ”basura”de lo que se estaba ejecutando anteriormente, en este caso el contexto de ejecución del kernel en el momento anterior, tiene base 0 limite mínimo (0x67) y protección nivel cero.

La tarea idle para tener una tarea de descanso del sistema operativo, con base en 0x0 limite 0x67 privilegio 0.

Después tenemos los descriptores de las tareas rick y morty (C137 Y D248) y las 20 tareas cronenberg ocupando limite mínimo para una tss.

3.2. pantalla (screen.c)

En este caso la pantalla esta dado por 80x50 píxeles donde cada píxel representa 8kb de memoria, el mundo Cronenberg ocupa 80x40 píxeles, tiene su propio segmento descrito anteriormente.

Las funciones ubicadas en screen.c se encargan básicamente de mostrar correctamente la información en pantalla. Entre ellas: screen init: se encarga de inicializar la interfaz visual con los indicadores de que tarea esta viva, para que se vea correctamente que hay 20 tareas si estan vivas o no , los cuadros de puntaje de los jugadores y la barra de fondo negra para distinguir visualmente el hud.

printScanCode: utilizada para detectar que tecla se apretó y decidir que hacer, si es un numero del 0 al 9 lo muestra en pantalla, si se toca la tecla Y se activa el modo debug, si ya estábamos en modo debug, se encarga de cerrar la ventana con los datos de los registros, restaurar la imagen y seguir con el juego matando la tarea que tiro la excepción.

imprimirDebug: imprime informacion para debugear de las tareas cuando generan una exepcion y se esta en modo debug

imprimirJugadores: se encarga de inicializar las posiciones en la pantalla de cada jugador y se encarga de mantener todos los iconos en su lugar correspondiente a medida que va progresando el juego dentro del mundo cronenberg

3.3. idt (idt.c)

En idt tenemos tenemos definidos los interrupt gate descriptors que en el caso de este tp solo usamos los interrupt gate. Tenemos definidas las interrupciones de 0 a 19 que son las excepciones del procesador. Luego tenemos las externas, la 32 que es la interrupción del clock y la 33 que es la del teclado. Por otro lado tenemos las entradas 137,138,139 correspondientes a las syscalls de usePortalGun, ImaRick y WhereIsMorty , estas ultimas están definidas con un DPL usuario ya que necesitamos que las tareas sean capaces de utilizarlas.

3.4. rutinas interrupciones (isr.asm)

por un lado tenemos las interrupciones de la 0 a la 19 que imprimen su numero en pantalla de interrupción y también tienen el funcionamiento del modo debug.
Modo debug: El modo debug se activa presionando la tecla Y, cuando esta se presiona se causa una interrupción de teclado y se activara una variable llamada modoDebugOn, si estamos en una Excepción, que lo sabremos porque activamos otra variable desde las isr0-19 tendremos el comportamiento que corresponde: salvaremos la pantalla, indicamos que ya no estamos en modo ExepcionDebug y mataremos a la tarea que causo la excepción.

la rutinas 32 y 33 corresponden a la rutina de atención de reloj y la de teclado.

isr32: Como se dijo previamente es la rutina de atención del clock, en esta lo que hacemos es preservar todos los registros,hacemos un call a pic finish1 para avisarle al pic que ya atendimos su interrupción y después llamamos a sched nextTask para que nos devuelva el selector de la próxima tarea a ejecutar.Si esta tarea es la misma que estábamos corriendo terminamos ahí y sino la cambiamos realizando un jmp far, esta rutina ademas de llama a resetearPortalesRick (implementada en isr.asm) y imprimirJugadores en screen.c que se encarga de imprimir en pantalla las 24 tareas y calcTareasMuertas (en game.c) se encarga de llevar la cuenta de cuantas tareas murieron para decidir si terminar el juego cuando se gane por puntaje.

isr33: preservamos los registros,llamamos picfinish1, leemos la tecla pulsada leyendo el puerto 0x60 y luego se la pasamos a nuestra función printScanCode para que pueda imprimirla en pantalla si corresponde.

Luego tenemos implementadas la rutina 137,138,139 que son para atender las syscalls:

isr137(usePortalGun):realizamos un pushad para preservar todos los registros y luego le pusheamos los parámetros a nuestra función auxiliar usePortalGun (implementada en sched.c), luego el resultado lo guardamos en memoria en un define dobleword hecho previamente para luego hacer popad y volver a cargar este valor en eax y poder retornarlo manteniendo los demás registros.

isr138:(iAmRick) realizamos un pushad para preservar todos los registros y luego llamamos a iAmRickAux (implementada en sched.c).

isr139(whereisMorty):similar a la rutina anterior hacemos un pushad y tenemos 2 doblewords reservadas para poder pasarle como parámetro a nuestra función auxiliar en sched.c (whereisMorty) y nos devuelva el resultado ahí, luego popad y movemos los resultados a eax y ebx.

Finalmente tenemos funciones auxiliares para las rutinas nextclock: el codigo encargado de actualizar el icono del reloj.

matarTareaAux: Esta función se encarga de realizar un jmp far a la tarea Idle, ya que cuando matamos a una tarea luego debemos ir a la tarea Idle.

ResetearPortalesAux: esta función es la que se encarga de que cada Rick solo sea capaz de usar un Portal por ciclo de clock.Tenemos un array donde cuando un Rick hace un portal este se setea en 1, en array[0] si es RickC137 o array[1] en caso contrario.Entonces cada vez que se ejecutara la rutina del Clock llamamos a esta función para setearlo a 0 y que Los Ricks sean capaces de volver a hacer portales.

3.5. game (game.c)

En game.c tenemos game init() que utilizamos para definir algunas estructuras que utilizaremos para ir progresando en el juego, les seteamos los valores a un arreglo con las posiciones de portales y otro arreglo que cuenta las puntuaciones. Luego tenemos la función endGame que es la encargada de terminar el juego.Esta función puede ser llamada desde cualquiera de los puntos de endGame establecidos, estos son: Que un Rick llegue a la máxima cantidad de puntos que puede llegar, por lo tanto el otro Rick no puede ganar. Que un Rick o Morty sean dominados, es decir les cambiaron el código. Que un Rick o Morty sea pisado. Y que un Rick o Morty causen una excepción. para cada una de las situaciones tenemos una función que es llamada en cada tipo distinto de engame y estas llaman a la función final endGame con el ganador correcto, ya que estas funciones pudrían ser llamadas desde el perdedor dependiendo el caso. las funciones son respectivamente: endGamePuntos, endGameMurioPisado, EndgameExepcion y endGameLoDominaron todas implementadas en game.c.

Tambien tenemos la funcion calcTareasMuertas() utilizada para decidir si terminar el juego = cuando puntaje de algun jugador +tareasmuertas=20 se finalizara el juego

3.6. paginacion (mmu.c)

Tenemos las distintas funciones correspondientes a las funciones relacionadas con paginación.

mmu init: para declarar el inicio de paginas libres

mmu zeropages: completa con ceros una pagina

mmu nextfrekernelpage: utilizada para devolver la siguiente pagina libre sin usar,

mmu mappage: para mapear una dirección física con una virtual, en un determinado cr3 definiendo nivel de privilegio (usuario/sistema) y si es readWrite o solo read se encarga de revisar si la page table requerida esta presente o no, en caso de no estarlo agrega una page table

mmu unmapPage: para des mapear una dirección virtual en un determinado cr3.

mmu initKernelDir: Inicializa el directorio de paginas y se inicializa la primer page directory entry con privilegios nivel usuario y readwrite para que estos parámetros dependan realmente de los atributos de cada pagina, en esta page table se inicializan 1024 paginas en supervisor y readwrite.

mmu initTaskDir: esta función se utiliza al inicializar una tarea, se encarga de mapear dentro del contexto de la tarea la información que necesita, el mapeo de 2 paginas de su propio código figure a partir de la dirección 0x08000000 en su propio contexto y el mapeo del kernel en las primeras 1024 paginas, osea mapeamos de 0 a 0x400000, para lograr hacer el mapeo del código figure 0x08000000 en su contexto debemos mapearlo en el cr3 de la tarea, en esa posición lineal con la posición física deseada en la que se almacenara, luego vamos al cr3 del kernel y hacemos un identity mapping usando la dirección física deseada del mundo cronenberg para luego teniendo esa dirección mapeada en el cr3 del kernel, podemos copiar el código de la tarea usando una dirección física del kernel y pegarla en la dirección física deseada como si fuese lineal con identity maping, finalmente des mapeamos ese identity maping en el kernel y logramos que en el contexto de la tarea su código figure en 0x08000000.

3.7. task state segment (tss.c)

Funciones relacionadas con el Task State Segment, definimos los descriptores de segmento de cada tarea en la gdt.

Tarea inicial: simplemente la definimos para que exista algún contexto del cual el scheduler pueda copiar al usarse por primera vez.

Tarea idle: la tarea que actualiza el reloj virtual en pantalla, posee el mismo cr3 que el kernel, se encuentra en 0x0001A000 y respeta los atributos pedidos en el enunciado (ejercicio 6.b), definimos sus registros de selectores de segmentos como los selectores de segmento de código y data de nivel 0 en base 0 (los primeros segmentos definidos en la gdt), el cr3 del kernel, su pila en 0x27000.

setTSSdescriptor: función auxiliar para setear dirección base de un descriptor tss en la gdt

tss init: les asignamos la dirección base al tss descriptor de la tarea idle e inicial en la gdt utilizando setTSSdescriptor, conseguimos las direcciones de las tareas cronenberg rick y morty utilizando la función getRandomPosition que asigna coordenadas (x,y) y estas las utilizamos para conseguir una dirección dentro del mundo cronenberg y posicionar las 24 tareas (las 2 rick, 2 morty y 20 cornenbergs, esto lo almacenamos en un arreglo , llamado direccionesCronenberg y luego llamamos a tss initCronenberg con la posición del arreglo de las direccionesCronenberg.

fillTSS: Se encarga de completar los datos de una tss, pasándole el índice de una tarea en la gdt, la direccion de la tarea en el kernel y la dirección física de la tarea cronenberg donde se quiere mapear , busca en el índice de la gdt la dirección base para calcular donde se ubica la tss, asigna el instructor pointer en 0x08000000, la pila en 0x08000000+2*pageSize, asigna a los registros de selector de segmento los segmentos de privilegio de nivel 3 y dpl=11, asigna los flags con interrupciones activadas, le asigna una pagina libre a la pila de nivel 0, y al segmento de nivel cero le selecciona el selector de segmento de la gdt que apunta al segmento de datos nivel cero, finalmente llama a mmu initTasDir con la dirección en el mundo cronenberg y la dirección de la tarea, para inicializar la tarea, hacer el mapeado, inicializar una page directory en el contexto de la tarea y devuelve este page directory para almacenarlo en el cr3 de la tss de la tarea.

Antes de definir tss initCronenberg se definen las direcciones de memoria para las 20 tss de las tareas cronenberg y un arreglo con estas direcciones.

tss initCronenberg: se encarga de cargar en la gdt los datos de cada descriptor de las 20 tareas cronenberg, les asigna limite 0x67, dpl=0, direccion base 0 entre otros atributos, luego utilizando setTSSdescriptor arregla la dirección base, para que cada tarea tenga su dirección base correctamente seleccionada, utiliza un arreglo (arregloCronenbergs) que indica la dirección física de las tss de cada tarea cronenberg. Luego llama a fillTSS para cada tarea cronenberg, dandole: el índice de la tarea en la gdt, la direccion de la tarea (0x18000, direccion de tarea La tarea cronenberg) y las direcciones físicas en el mundo cronenberg de cada una y se va a encargar de cargarles la TSS a cada una e inicializarles su cr3, y dentro de su contexto a cada tarea mapear el kernel en las primeras 1024 paginas y su propio código en la dirección 0x08000000.

3.8. scheduler(sched.c)

schedinit: No se usa.

schednextask: se encarga de seleccionar la siguiente tarea en la gdt, para esto tenemos un vector de índice de tareas con las 24 tareas (20 cronenberg y los 2 pares de rick y morty), se fija que la tarea este activa, en caso de no estarlo busca una tarea siguiente en la lista, finalmente devuelve el índice de la gdt de la próxima tarea activa.

matarTareas: marca una tarea como muerta, en el arreglo que utilizamos para distinguir si una tarea esta viva o muerta, y luego llama a matarTareaAux definida en isr.asm esta función auxiliar esta explicada en sección rutinas de interrupción.

Funciones auxiliares de rutinas de interrupción:

WhereIsMortAux, se encarga de conseguir el selector de segmento actual con la función rtr para saber desde que tarea se llamo a la interrupción, luego sabiendo que tarea llamo a WhereIsMortAux shifteamos 3 bits a derecha el selector para que sea un índice de la gdt, restamos 15, al restarle 15 (necesitamos restar 15 ya que en la gdt hay 15 elementos antes de las tareas) el índice ahora se utiliza en un arreglo

que indica las posiciones de las tareas en el mundo cronenberg, con estas direcciones ya podemos calcular el eje x y el eje y de nuestro rick y morty y calcular, el desplazamiento del eje x y el del eje y entre ambos.

iamRickAux: se encarga de comparar el selector de segmento actual usando rtr, comparándolo con los selectores de segmentos de cada tarea que esta inicializada en la gdt, para esto utilizamos un arreglo llamado indicesGDTtareas que contiene esa información, habiendo encontrado que estamos en un índice de tarea que corresponde a nuestro rtr, verificamos si el código es 0xc137 o 0xd248 y sumamos punta je al equipo de la dimensión 0xc137 o 0xd248 dependiendo cual era el código respectivamente.

usePortalGun: primero se encarga de encontrar que tarea llamo a la función, utilizando indiceGDT-tareas, con esto se ya podremos saber en que ubicación esta la tarea que llamo al portal, si es una tarea rick, primero revisa tener disponible el portal, utilizando el vector usadoPortalRickDisMorty que indica si aun no esta disponible su portal, en caso contrario se utiliza el portal y calcula los ejes x y del rick y morty actual. en caso de ser un morty revisa si esta disponible el uso del portal, en caso de no haber pasado 10 turnos se saldrá de la función, en caso contrario calcula las posiciones x y de morty, si soy una tarea cronenberg se mata a esa tarea y se sale de la funcion. Se calcula la dirección final a la que iran a parar los ricks y mortys considerando el desplazamiento pedido para poner un portal, se calculan la ubicación en memoria del portal de rick y el de morty. Verificamos si el parámetro cross=true si vamos a pisar una tarea o no, vemos si pisamos alguna tarea que no fue la que llamo el portal, si algún portal, el rick o el morty pisan una tarea, chequeamos si pisamos a algún morty o rick con ese portal, en tal caso llamamos a endgameMurioPisado, que se encarga de terminar el juego, sino, mata a la tarea cronenberg. separamos en casos:

Si me llamo un rick: si cros=0 y withmorty=0 mapeamos la dirección física donde apunta el portal a 0x08002000 dir virtual, y a la dirección del portal rick mas el tamaño de una pagina, eso lo mapeamos con la dirección física de 0x08002000, en el cr3 del kernel.

Si cross=0 withmorty es 1 hacemos lo mismo, y luego conseguimos el cr3 de la tarea morty usando la gdt y la tss, luego mapeamos la dirección lineal 0x08002000 con la dirección del portal morty en el cr3 de morty y la dirección lineal 0x08002000+PAGE SIZE con la dirección física direcPortalMorty+PAGE SIZE.

Si cross=0 withmorty=0 mapeo con identity maping la dirección del portal rick y luego la dirección de portal lineal direcPortalRick+page SIZE la mapeo con la dirección física direcPortalRickk+page SIZE en el cr3 del kernel.

Ahora copiamos lo que esta en la dirección del portal de rick a 0x08000000, mapeo en el cr3 del kernel dirección lineal 0x08000000 con la direccion física = dirección del portal rick y mapeo 0x08000000 con dirección física= dirección del portal rick+PAGE SIZE y luego des mapeo la dirección lineal del portal rick y la dirección lineal del portal rick+ PAGE SIZE, y posiciono la tarea actual en la dirección del portal rick.

Si cross=1 y withmorty=1, identity mapping con la dirección del portal rick y lo mismo con dirección portal rick+ PAGE SIZE, copiamos en la dirección del portalrick lo que esta en 0x08000000, mapeo la dirección lineal 0x08000000 con la dirección física= portal rick lo mismo con ambas sumadas a PAGE SIZE y des mapeamos el mapeo auxiliar, des mapeamos la dirección lineal del portal rick y la dirección mas una pagina siguiente. Ahora nos encargamos con morty, conseguimos el cr3 de morty similar a como lo hicimos anteriormente en una de las anteriores opciones, mapeamos con identity mapping la dirección del portal morty y la de la pagina siguiente, copiamos en la dirección del portal el contenido de la dirección 0x08000000, mapeamos en el cr3 de morty la direccion 0x08000000 con la dirección física del portal morty y lo mismo con ambas direcciones + PAGE SIZE, desmapeamos la dirección del portal morty y la pagina siguiente cargamos el cr3 de rick y actualizamos la posición de rick con su portal y morty con su portal.

Si me llamo un morty: si cross=1, identity maping de dirección portal morty mas la siguiente pagina, copiamos en la dirección del portal de morty el contenido de la dirección 0x08000000 re mapeamos la dirección lineal 0x08000000 con la dirección del portal morty y lo mismo para ambas direcciones + PAGE SIZE desmapeamos la dirección portal morty y la dirección portal morty+ PAGE SIZE y actualizamos la posición del portla morty.

Si cros=0 MAPEAMOS LA DIRECCION física PORTAL MORTY a lineal 0x08002000 y la siguiente pagina en ambas y actualizamos la posición del portal de morty.

4. Respuesta de los Ejercicios del enunciado

4.1. Ejercicio1

1. Se encuentra en gdt.c son los primeros 4 segmentos definidos por nosotros (a partir de de la posición 8) especificadas como en el enunciado.
2. Activamos a20 y cargamos la gdt y activamos el bit PE para poder pasar a modo protegido haciendo un jmp far, todo esto en kernel.asm.
3. Se encuentra en gdt.c como la 5 entrada definida por nosotros.
4. Se encuentra en kernel.asm, básicamente es un ciclo que pinta toda la pantalla de verde.

4.2. Ejercicio2

1. Se encuentra en idt.c, definimos las entradas de 0 a 19, 32, 33 todas usando una macro con atributos 8e00.

   Mientras que las interrupciones 137,138,139 las definimos con los atributos EE00
2. En kernel.asm cargamos la idt y en su momento la testeamos haciendo una división por cero, actualmente no dividimos por cero para no interrumpir al kernel.

4.3. Ejercicio3

1. Las entradas en la idt para la interrupción de reloj y teclado son las 32, 33 y las entradas 137, 138 , 139 inicializadas en idt.c
2. En isr.asm se encuentra la rutina asociada a la interrupción del reloj, que cumple con lo pedido de actualizar la animación del clock.
3. En isr.asm en la rutina de teclado se llama a printScanCode definida en screen.c, se encarga de distinguir las teclas del 0 al 9 y imprimir en pantalla la tecla correspondiente.
4. Las rutinas asociadas a las interrupciones 137, 138 y 139 se encuentran en isr.asm estas rutinas fueron modificadas por un ejercicio próximo.

4.4. Ejercicio4

1. mmu initKernelDir se encuentra en mmu.c, esta función esta explicada anterior mente en la sección de mmu, inicializa un page directory y un page table, el page table con user y read write, y las carga todas las paginas posibles en esa page table (1024 paginas de 4k) con privilegio supervisor y readwrite
2. En kernel.asm corremos la funciones mmu init, Que define la primer pagina libre, mmuinitKernelDir Y CARGAMOS EL CR3.
3. Se llama a libretas al final de todo en kernel.asm, se encuentra definida al final de screen.c.

4.5. Ejercicio5

1. MMU init se encuentra en mmu.c, en nuestro caso lo único que hace es seleccionar la primer pagina disponible.
2. mmu mapPage: se encuentra en mmu.c, mapea una dirección física con una virtual, dado un cr3 y los privilegios read write o user supervisor.

   mmu unmapPage: Des mapea una dirección virtual dado la dirección virtual y el cr3.
3. mmu initTaskDir se encuentra en mmu.c, esta explicada anteriormente en la sección de paginación.
4. Ya no se encuentra implementado tal como dice el enunciado.

4.6. Ejercicio6

1. Definimos en gdt.c la entrada de la tarea inicial e idle, con tamaño mínimo, bit indicando tss, privilegio 0 base=0.
2. En tss.c completamos la tss de la tarea idle .
3. En tss.c completamos la entrada de la gdt correspondiente a tarea inicial utilizando la función tss init.
4. En tss.c completamos la entrada de la gdt correspondiente a tarea idle utilizando la función tss init.
5. El código para intercambiar tarea se encuentra en kernel.asm linea 160 se encarga de hacer un jmp a la tarea idle desde la inicial.
6. La función que completa la tss es FILTSS se encuentra en (tss.c) explicada en la sección tss.

4.7. Ejercicio7

1. Utilizamos la función sched init en sched.c en principio no inicializamos nada, definimos INDICETAREAS=-1.
2. sched nextTask se encuentra en sched.c esta explicada en la sección de scheduler.
3. Las rutinas de interrupción 137, 138 y 139 se encuentran isr.asm explicadas en la sección de rutinas de interrupción.
4. sched nextTask se encuentra en sched.c y se encuentra explicada en la seccion de scheduler.
5. Agregamos un call a matarTarea con el índice correcto para que se encargue de matar a esta y luego salte a la tarea Idle.
6. Explicado en la sección isr.asm.