DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 3

System Programming - Zombie defense

Organización del computador 2

Primer Cuatrimestre de 2017

Grupo Estrellitas

Integrante	LU	Correo electrónico
Hofmann, Federico	745/14	federico2102@gmail.com
Lasso, Andrés	714/14	lassoandres2@gmail.com
Berríos Verboven, Nicolas	46/12	nbverboven@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (++54 +11) 4576-3300

http://www.exactas.uba.ar

Índice

Parte I
Introducción

El objetivo de este trabajo fue realizar una primera aproximación a la programación de sistemas. El mismo debe poder correr 16 tareas (zombies) de forma concurrente a nivel usuario y será capaz de desalojarlas al producirse cualquier tipo de problemas con las mismas, así como también se las podrá cargar al sistema por medio del uso del teclado.

Se trataron temas como inicialización del procesador, gestión de memoria, manejo de interrupciones, mecanismos de protección y gestión de tareas. Es nuestro trabajo entender a fondo cada uno de esos temas para poder implementarlos y luego describirlos de manera detallada.

Parte II
Desarrollo del trabajo práctico

1. Segmentación y pasaje a modo protegido

1.1. Tabla de descriptores globales (GDT para los pibes)

El primer paso fue agregar en la GDT los descriptores para segmentos de código y datos de nivel 0 y 3 -es decir, con niveles de privilegio de kernel y usuario- que permitieran direccionar los primeros 623 MB de la memoria, siguiendo un modelo de segmentación *flat*. En todos los casos, la base apunta al comienzo de la memoria, es decir la dirección 0x00000000, y el límite es 0x26EFF. Este valor se debe a que

$$\begin{aligned} 623 \text{ MB} &= 623 * 1024 \text{ KB} = 623 * 2^{10} \text{ KB} \\ &= 623 * 2^8 * 2^2 \text{ KB} = 623 * 256 * 4 \text{ KB} \\ &= 159488 * 4 \text{ KB} \\ &= 159488 \text{ incrementos de 4 KB.} \end{aligned}$$

Ahora bien, $(159488)_{10} = (0x26F00)_{16}$ y, si se considera que cada segmento comienza en la dirección 0x00000000, entonces el máximo offset -en bytes- será 0x26EFFFFF[1] . Como no es posible representar este valor en el campo límite del descriptor, fue necesario poner el bit de granularidad en 1 y dejar expresado el valor como cantidad de incrementos de 4 KB. Se definió también un segmento de código de nivel 0 para el área de la memoria de video correspondiente a la pantalla, con inicio en la dirección 0xB8000 y un tamaño de $80 * 50 * 2$ *bytes* = 8000 *bytes*.

Tabla 1: aaaaaaa

Índice	Base	Límite	P	DPL	S	D/B	L	G	Tipo
8	0x00000000	0x26EFF	1	0	1	1	0	1	0x08 (code, execute-only)
9	0x00000000	0x26EFF	1	3	1	1	0	1	0x08 (code, execute-only)
10	0x00000000	0x26EFF	1	0	1	1	0	1	0x02 (data, read/write)
11	0x00000000	0x26EFF	1	3	1	1	0	1	0x02 (data, read/write)
12	0x000B8000	0x01F3F	1	0	1	1	0	0	0x02 (data, read/write)

1.2. Modo protegido

Para pasar a modo protegido, fue necesario habilitar el pin A20 del procesador (se utilizó la instrución `habilitarA20` provista por la cátedra) para poder direccionar más de 1 MB, a diferencia que lo que sucede en modo real. Luego, se cargó la dirección base de la GDT en el registro GDTR

[1] Porque 4 KB = $4 * 1024$ *bytes* = 4096 *bytes* = 0x1000 *bytes*. Entonces, 0x26F00 * 4 KB = 0x26F00 * 0x1000 *bytes* = $0x26F00000$ *bytes*. Si a ese valor le restamos 1, se obtiene el máximo offset para un segmento de 623 MB si se comienza a contar desde el byte 0.

mediante la instrucción `lgdt`, se puso en 1 el bit menos significativo del registro `CR0` para activar el modo protegido y se realizó un salto a la primera instrucción utilizando el selector de segmento de código de nivel 0.

Una vez en modo protegido, se cargaron los registros de segmento de la siguiente manera:

- `ds`, `es`, `gs`, `ss`: selector de datos de nivel 0
- `fs`: selector de video (datos de nivel 0).

Además, se estableció la base de la pila del kernel en la dirección 0x27000 cargando la misma en los registros `esp` y `ebp`.

2. Manejo de excepciones

Se definieron entradas en la IDT para atender las 20 excepciones que posee por defecto el procesador por medio de un macro provisto por la cátedra modificado -como se ve en la Figura 1- para incluir un argumento que permitiera indicar el tipo de descriptor al que corresponde cada elemento de la tabla.

```
#define IDT_ENTRY(numero, tipo_puerta)                                                  \
    idt[numero].offset_0_15 = (unsigned short) ((unsigned int)(&_isr ## numero) & (unsigned int) 0xFFFF); \
    idt[numero].segsel = (unsigned short) 0x0040;                                       \
    idt[numero].attr = (unsigned short) tipo_puerta;                                    \
    idt[numero].offset_16_31 = (unsigned short) ((unsigned int)(&_isr ## numero) >> 16 & (unsigned int) 0xFFFF);
```

Figura 1: Extracto del achivo `idt.c` mostrando el código del macro utilizado para definir las entradas de la IDT.

En el caso puntual de las excepciones se utiliza un descriptor de puerta de interrupción con un nivel de privilegio de kernel pero, como se verá más adelante[2] , podría pasar que hiciera falta una entrada que tuviera una variación en este campo, y esta forma resulta cómoda para trasladar esto al código.

El selector de segmento utilizado en todos los casos corresponde al índice 8 de la GDT, es decir a un descriptor de segmento de código de nivel 0, lo que tiene sentido pues las excepciones no serán atendidas por los usuarios sino por el kernel.

El campo `attr` se compone de 4 bytes de los cuales los 2 menos significativos son 0. El resto se completó de acuerdo a los requerimientos de una puerta de interrupción de 32 bits, presente y con un DPL = 0. El resultado es el valor 0x8E00.

El offset es relativo al segmento apuntado por `segsel` y apunta a la rutina de atención de la excepción, definida en el archivo `isr.asm`. Éstas consistían en imprimir por pantalla qué error se había producido y detener la ejecución. Posteriormente fueron modificadas, como se detalla en la sección 7.

Se inicializó la IDT con la función `idt_inicializar` y se cargó su dirección en el registro `idtr` con la instrución `lidt`.

Algoritmo 1 Pseudocódigo de la rutina *idt_inicializar*.

1: $interrupt\_gate\_kernel \leftarrow$ 0x8E00
2: **for** $i = 0$ **to** 19 **do**
3: $\quad IDT\_ENTRY(i, interrupt\_gate\_kernel)$
4: **end for**

3. Paginación y memory mapping (kernel)

Los primeros 4 MB de la memoria están comprendidos entre las direcciones físicas 0x00000000 y 0x003FFFFF. Si ahora se consideran estos valores como direcciónes virtuales puede verse que el índice en el directorio de tablas (bits 31:22) siempre es 0, mientras que los de la tabla de páginas (bits 21:12) van desde el 0 hasta el 1023 (0x3FF en hexadecimal). Con esta información puede concluirse que, para mapear este área de la memoria, es necesaria una sola *page table* cuya dirección física será almacenada en el índice 0 del *page directory* del kernel.

[2]Los descriptores que corresponden a los servivios del sistema, por ejemplo, son puertas de interrupción; sin embargo, el nivel de privilegio debe ser tal que permita que las tareas puedan utilizarlos. Más información en la sección

Se sabía previamente que el directorio de tablas del kernel comenzaba en la dirección 0x00027000, mientras que la tabla de páginas lo hacía en la dirección 0x00028000.

Se completó el índice 0 del directorio de tablas con el valor 0x00028003; es decir, se indicó que la tabla de páginas del kernel poseía un nivel de privilegio de supervisor, estaba presente y era de lectura/escritura (los otros atributos no resultaban relevantes a los efectos de este trabajo práctico, por lo que quedaron en 0). El resto de las entradas se puso en 0.

Para obtener un mapeo de identidad de los primeros 4 MB, las entradas de la tabla de páginas del kernel se completaron con las direcciones de inicio de los marcos de página (cuyos valores iban desde 0x0 hasta 0x3FF) con los mismos atributos que los asignados a la primera entrada del directorio de páginas.

Para finalizar, se cargó la base del *page directory* del kernel en el registro `CR3` y se habilitó paginación seteando el bit más significativo de `CR0` en 1.

4. Memory mapping (tareas)

Una de las cosas que puede observarse leyendo las secciones anteriores es que tanto la pila del kernel como su directorio de tablas y su tabla de páginas poseen direcciones predefinidas. Para realizar el mapeo de memoria de las tareas, se declaró una variable global que, cuando se inicializa la unidad de manejo de memoria, comienza conteniendo la dirección 0x00100000. Se definió también una función `proxima_pagina_fisica_libre` que devuelve el valor almacenado en la mencionada variable e incrementa su valor en 0x1000.

Algoritmo 2 Mapear una página

Input: Dirección virtual, dirección física, el contenido de `CR3`, el nivel de privilegio que tendrá la pagina a mapear y si esta será de lectura, escritura o ambas

1: Obtengo los índices del *page directory* y de la *page table* a partir de la dirección virtual
2: Obtengo la dirección del *page directory* con `CR3`
3: **if** no existe la entrada del *page directory* dada por el índice obtenido **then**
4: Pido una página de memoria libre
5: Pongo el bit de presente en la entrada del *page directory* en 1
6: Completo la entrada del *page directory* con la dirección de dicha página, el nivel de privilegio y *r/w* que se pasan como parámetro
7: Inicializo todas las entradas de la nueva *page table* en 0
8: **end if**
9: Completo la entrada correspondiente de la tabla de páginas con la dirección fisica, el nivel de privilegio y *r/w* que se pasan como parámetro
10: Pongo el bit de presente en la entrada de la *page table* en 1
11: Invalido la cache de traducción de direcciones

Para construir el mapa de memoria de una tarea se definió la función `mmu_inicializar_dir_zombie`, que toma como argumentos:

- la dirección física en la que se encuentra el código del zombie en cuestión,
- el jugador que lanzó la tarea,
- su respectivo `CR3`,
- las ubicación del zombie en el mapa (como coordenadas x e y).

En primer lugar, se mapearon los primeros 4 MB con *identity mapping* siguiendo un procedimiento análogo al descripto en la sección 3. Luego, se procedió a realizar el mapeo de las direcciones virtuales de la tarea a las pociciones del mapa (o, mejor dicho, al área de la memoria que vendría a representarlo). Para calcular las direcciones de esas páginas, se utilizó una estructura llamada `map_tile` definida de forma tal que ocupara exactamente 4096 bytes; así, fue posible operar interpretando el mapa/memoria como si fuera una matriz de páginas de 4 KB, de 44 filas y 78 columnas, con inicio en la dirección 0x00400000, abstrayéndose de esta forma del cálculo manual de las direcciones.[3]

[3]Por ejemplo, si se deseara mapear las páginas de un zombie que se encuentra en una posición del mapa (x, y), las

El mapeo de las páginas de las tareas se realizó con privilegio de usuario (excepto la parte del kernel, a la que se le asignó un privilegio de supervisor) y atributos de lectura/escritura siguiendo el Algoritmo 2. Como último paso, se copió el código del zombie en la posición indicada por las coordenadas x e y que recibió la función `mmu_inicializar_dir_zombie`.

Algoritmo 3 Desmapear una página

Input: Dirección virtual que se desea desmapear, contenido de `CR3`
Requiere: La dirección virtual debe contener indices correspondientes a entradas presentes
1: Obtengo los índices del *page directory* y de la *page table* a partir de la dirección virtual
2: Obtengo la dirección del *page directory* con `CR3`
3: Accedo a la *page table* con el índice correspondiente del *page directory*
4: Pongo el bit de presente en la entrada de la *page table* en 0
5: Invalido la cache de traducción de direcciones

5. Interrupciones externas

Lo primero que se hizo para tratar con estas interrupciones fue agregar a la IDT explicada en la sección *Manejo de excepciones* entradas para los descriptores del clock, teclado y syscall 0x66. Estas entradas son la 32, 33 y 102 con privilegios de kernel las dos primeras y de usuario la syscall 0x66.

Posteriormente se definieron en el archivo *isr.asm* las rutinas de atención a las tres interrupciones referenciadas por los descriptores recién mencionados.

5.1. Clock

La idea principal de esta interrupción es que se muestre por pantalla un reloj moviéndose a cada ciclo del clock. Para esto fue necesario realizar los siguientes pasos:

- Salvar la información de los registros generales y de los eflags mediante las instrucciones *pushad* y $pushfd$
- Avisar al pic que la interrupción fue atendida (las rutinas del pic nos fueron dadas previamente)
- Llamar a la función $proximo_reloj$, la cual es la encargada de hacer que el reloj en pantalla se mueva una posición (esta función también nos fue dada de antemano)
- Restaurar el contenido de los registros de propósito general y de los eflags mediante las instrucciones popad y popfd
- Retornar mediante la instrucción iret.

5.2. Teclado

En principio el objetivo de esta interrupción es mostrar por pantalla la tecla que se esta apretando, si es que esta es una de las que se utilizan en el juego. Por el momento, esta rutina realiza lo siguiente:

- Salva los eflags y registros de propósito general mediante *pushad* y $pushfd$
- Limpia el contenido el registro *eax*
- Carga en la parte baja de este registro el valor de la tecla presionada
- Hace *push* del registro para pasarlo como parámetro a la función en C que se encargara de realizar la acción deseada
- Hace *call* a dicha función
- Aumenta el *esp* en 4

direcciones de las posiciones pueden obtenerse simplemente mediante las instrucciónes `&mapa[x][y]`, `&mapa[x+1][y]`, etc. en lugar de tener que sumar "a mano" de a 4 KB desde la dirección 0x00400000 (suponiendo que *mapa* es una matriz de elementos de 4 KB que comienza en esta última dirección).

- Llama a pic_fin_int para avisar que se atendió la interrupción
- Recupera el estado de los registros de propósito genera y los eflags
- Vuelve a la rutina principal con *iret*

No se describirá en esta sección a la función encargada de mostrar por pantalla la tecla que se presionó ya que esta fue modificada posteriormente para realizar diferentes acciones acordes a las funcionalidades del juego.

5.3. Syscall 0x66

Por lo pronto, esta interrupción simplemente mueve al registro *eax* el valor 0x42, salvando los registros de propósito general y los eflags previamente y volviendo a su estado original al final y retorna con *iret* del mismo modo que se hizo en los dos casos anteriores. Esta rutina será modificada en la sección 7 al igual que las otras dos para atender a funcionalidades esperadas en el juego.

6. Segmentos de estado de tareas (TSS)

Antes de poder comenzar a correr las tareas es necesario preparar algún tipo de mecanismo para almacenar todos los datos necesarios del estado de cada tarea para de este modo poder alternar la ejecución de las mismas sin perder su información. La forma de conseguir esto es mediante los TSS (Task State Segment).

Lo primero que debimos hacer en este sentido fue definir en la GDT entradas para los descriptores de TSS: uno para la tarea inicial, que permite que se realice el primer intercambio, otro para la tarea *Idle* y uno para cada una de las tareas que fuera posible ejecutar concurrentemente cada jugador. El formato de dichas entradas es el siguiente:

- El limite es 0x67 ya que el mínimo tamaño que puede tener una TSS para almacenar todo el contexto de una tarea es 0x68
- El campo *tipo* lo completamos con el valor 9b ya que, con s=0, corresponde a un TSS de 32 bits disponible
- El bit de presente se pone en 1
- El bit d/b en 1 indicando que es un segmento de 32 bits

Todos los campos no mencionados se encuentran en 0.

Una vez hecho esto, se procedió a completar la entrada de TSS de la tarea Idle en el archivo TSS.c con la información detallada en la consigna. Para esto se utilizo la funcion $tss\_inicializar\_idle$ que primero actualiza el comienzo del TSS de la tarea idle en su descriptor de la GDT y luego completa los campos de la TSS en cuestión de la siguiente forma:

- Esta tarea debía compartir el page directory con el kernel, por lo que su $cr3$ se inicializó con esa dirección
- También debía compartir la pila con el kernel, por lo que los registros *ebp* y *esp* de esta tarea apuntan a la pila del kernel
- Ademas, el *eip* se configuro con el valor 0x00016000 como se indico en el enunciado
- Los eflags se setean en 0x00000202; esto es, con las interrupciones activas
- Para el segmento de código *cs* se le asigno el selector de segmento de nivel 0
- Para ds, es, gs, fs y ss el selector de datos de nivel 0
- Por ultimo, los registros *eax*, *ebx*, *ecx* y *edx* se inicializaron en 0.

Para completar estos datos, se nos facilito una estructura con los campos mencionados en el listado anterior en el archivo TSS.h.

Una vez hecho esto procedimos a crear una función encargada de inicializar las TSS de las tareas zombies. Esta función lo primero que hace es obtener un puntero a una TSS libre y luego de obtenerla completa sus campos de la siguiente manera:

- El $cr3$ se obtiene a partir de lo que retorna la función *mmu_inicializar_dir_zombie*
- El *eip* se configura con la dirección virtual del mapa, la cual es una variable global definida como 0x08000000
- El *ebp* y *esp* inician en la posición 0x08000000 + 0x1000, de este modo la pila crecerá desde el final de la pagina de la tarea
- Eflags = 0x00000202
- El *cs* sera un selector de código de nivel 3 declarado de manera global y que referencia a su respectiva entrada en la GDT
- ds, es, gs, fs y ss serán el selector de datos de nivel 3 de la misma forma que en el ítem anterior
- $esp0$ sera la pila de nivel 0, y para esto se usa el resultado de la llamada a la función $mmu\_proxima\_pagina\_fisica\_libre$+0x1000 para que decrezca desde el final de esta nueva pagina.
- El $ss0$ sera un selector de datos de nivel 0 declarado de forma global y referenciando al de la GDT
- Los registros eax, ebx, ecx y edx se inicializan en 0.

También se implemento la función *tss_inicializar*, la cual se encarga de actualizar el comienzo de la TSS de las tareas en sus respectivos descriptores en la GDT.

Algoritmo 4 Pseudocódigo de la rutina *tss_inicializar*

```
1: arreglo_tss_A[8]
2: arreglo_tss_B[8]
3: tss_inicial                                          ▷ Variables globales
4: gdt[indice_tarea_inicial].base ← &tss_inicial
5: for i ← 15 to 22 do                  ▷ Indices de la gdt de las tareas del jugador A
6:     gdt[i].base ← &arreglo_tss_A[i]
7: end for
8: for i ← 23 to 30 do                  ▷ Indices de la gdt de las tareas del jugador B
9:     gdt[i].base ← &arreglo_tss_B[i]
10: end for
```

7. A jugar con zombies

Para la ultima parte de este trabajo se debieron implementar funciones que pongan en funcionamiento el *scheduler*, encargado de asignar tiempo y recursos del sistema a cada tarea para que estas puedan correr de forma concurrente. Lo primero que se hizo fue crear una serie de estructuras que nos permitieran obtener y actualizar información sobre las tareas, los jugadores e información general del juego:

- `Jugador`, cuyos campos son:
  - `Zombies_restantes`: Cantidad de zombies que el jugador actual todavía puede lanzar.
  - `Posicion_x`: Posición en el eje x (columnas) del jugador.
  - `Posicion_y`: Posición en el eje y (filas) del jugador.
  - `Proximo_zombie_a_lanzar`: Tipo de zombie seleccionado. Puede ser clérigo, mago o guerrero.
  - `Puntaje_actual`: Puntaje del jugador actual hasta el momento
- `Task_info`, que contiene información sobre las tareas (zombies):
  - `esta_activa`: Informa si la tarea zombie fue lanzada o no.

  - `selector_tss`: Contendrá el selector de tss de la tarea
  - `z_tipo`: Tipo de zombie que es la tarea. Mago, clérigo o guerrero.
  - `z_posicion_x`: Posición en el eje x de la tarea actual
  - `z_posicion_y`: Posición en el eje y de la tarea actual.
  - `z_reloj`: Para pintar los relojes de las tareas. Se actualizan con cada tick de reloj.

- `Info_juego`, que refleja información mas global sobre el juego:
  - `zombies_disponibles[3]`: Mago, clérigo o guerrero.
  - `jugador_de_turno`: 0 si es el jugador A, 1 si es el jugador B.
  - `jugador_A`: Información del jugador A.
  - `jugador_B`: Información del jugador B.
  - `tarea_actual_A`: Tipo de zombie seleccionado del jugador A.
  - `tarea_actual_B`: Tipo de zombie seleccionado del jugador B.
  - `tareasA[CANT_ZOMBIS]`: Zombies del jugador A.
  - `tareasB[CANT_ZOMBIS]`: Zombies del jugador B.

Teniendo ya estas estructuras definidas se procedió a crear una función para inicializar el scheduler. La misma se encarga de inicializar las estructuras recién explicadas con los valores correspondientes:

- Se crea una variable global de tipo info_juego y se procede a completar sus campos dentro de esta función:
  - Para el jugador A y el jugador B se indica las posiciones en las que comienzan, se definen sus puntajes y sus próximos zombies a lanzar.
  - Se selecciona al jugar A como jugador de turno y se establecen los zombies disponibles.
  - También se recorre los indices en la GDT de los descriptores de las tareas del jugador A y del jugador B y se marca como inactivas todas sus tareas y se cargan los selectores de TSS de las mismas.
  - Finalmente se inicializa la pantalla del juego con toda esta información.

El siguiente paso consistió en implementar una función que devuelva el indice de la TSS de la siguiente tarea activa a ejecutarse. Esto servirá después para realizar el cambio de tareas. Esta función lo que hace es obtener la lista de tareas del jugador de turno (el jugador de turno se actualiza cada vez que se llama a esta función) y buscar una tarea activa distinta de la actual si la hay, y sino la tarea actual y devolver el selector de TSS de dicha tarea. En caso de no haber ninguna tarea activa se retorna 0.

Para implementar el cambio de tareas lo que se hace es agregar esta ultima función a la rutina de atención del clock. De este modo, a cada tick del clock se buscara la próxima tarea activa del jugador que no sea el actual y se procederá a atenderla. En caso de que el otro jugador no tenga ninguna tarea activa entonces se elegirá la próxima tarea activa del mismo jugador. Si solo tiene una tarea activa se continua ejecutando la misma.

La siguiente modificación realizada fue a la rutina de atención a la syscall 0x66. En esta se introdujo un llamado a la función *game_move_current_zombi*, la cual se encarga de mover al zombie lanzado según indique el código de la tarea. Para lograrlo, esta función desmapea las paginas del zombie y las vuelve a mapear en la posición indicada. Finalmente imprime una x en la nueva posición del zombie en el mapa. Esta rutina es llamada tantas veces como el código de la tarea en movimiento lo indique.

También hubo que determinar procedimientos para desalojar tareas cuando estas produjeran alguna excepción. Para esto se creo la función *sched_desalojar_tarea_actual* la cual es llamada desde la rutina de atención a excepciones del procesador y funciona de la siguiente manera:

- Busca al jugador cuya tarea produjo la excepción (infoJuego.jugador_de_turno).
- Marca a su tarea actual como inactiva.
- Actualiza los putajes de los jugadores si hace falta (si la posicion_x de la tarea era 0 o 77).

Por último, se debió implementar un mecanismo de debugging para indicar por pantalla la razón de desalojo de la tarea. Este mecanismo se activa presionando la tecla y mientras la tarea/s esta corriendo y en el momento en que alguna tarea sea desalojada se muestra la pantalla recién mencionada. Para implementar este mecanismo se realizaron los siguientes pasos:

- Se crearon dos variables globales que indican si el modo debugging esta activado y si la pantalla de debugging esta activada respectivamente.
- La rutina de atención al teclado mira si se presiono la tecla y.
  - En caso afirmativo se fija si el modo debugging ya estaba activado
    - De ser así se fija si ya estaba mostrando la pantalla de debugging
      - Si se estaba mostrando entonces deja de mostrarla, se desactiva el modo debugging y se vuelve a la ultima versión de la pantalla
      - Sino se sale de modo debugging y se borra el mensaje que informa que dicho modo esta activado.
    - Si no estaba activado el modo debugging, se lo activa y se muestra el mensaje *mododebuggingactivado*
  - En caso negativo se procede a llamar a la función *game_jugador_mover*, la cual se encarga de mover hacia arriba o hacia abajo al zombie del jugador correspondiente, cambiar de zombie o lanzar un zombie seleccionado según corresponda.
- La rutina de atención al clock mira si la pantalla de debugging esta activa y en ese caso salta a la tarea *Idle*, caso contrario continua con el intercambio de tareas.
- La rutina de atención a interrupciones del procesador mira si el modo debugging esta activo.
  - Caso afirmativo, se muestra la pantalla de debugging con la información del estado de los registros de la tarea desalojada.
  - Sino, se muestra en la parte superior izquierda de la pantalla la excepción producida y se desaloja a la tarea que la produjo.

Parte III
Conclusiones

En ciertas oportunidades a lo largo de la realización del trabajo práctico se presentó la posibilidad de elegir entre implementar el código en C o en assembly. Un caso fue el de pintar la pantalla con la interfaz del juego. Aquí se optó por trabajar en código ensamblador, puesto que no revestía demasiada complejidad y se trataba de una rutina relativamente corta. Sin embargo, cuando se implementaron las rutinas de atención de interrupciones externas, especialmente la del teclado, se buscó implementar la mayor cantidad de funcionalidades posible en C dado que no resultaba trivial y un lenguaje de más alto nivel facilitaba la escritura del código y su posterior comprensión. Esto trajo como contrapartida que ciertas operaciones se ralentizaran. Un caso puntual fue el de las interrupciones del teclado; la función encargada de atenderlas (menos la de la tecla `Y`) estaba escrita en C pero estaba implementada como un *switch/case*, lo que hacía que encontrar la intrucción a ejecutar dada una tecla fuera lento. Otra posibilidad es que la lentitud se debiera a los algoritmos utilizados, independientemente del lenguaje.

Surgió un problema al implementar el modo debug. Si se está en ese modo al morir el primer zombie, el lugar de imprimirse la pantalla con la información de la tarea desalojada, lo que aparece en su lugar es una sección pintada de negro. Es importante destacar que si se presiona la tecla `Y` en esta situación el sistema se comporta de forma normal restaurando la pantalla correctamente; también debe aclararse que el error no se vuelve a producir en subsiguientes muertes de otros zombies. Aún más, si la primera muerte de un zombie se produce no estando activo el modo debug, no se detectan problemas durante el resto de la ejecución. Durante el proceso de búsqueda de la causa del mal funcionamiento se vio que si se comentaba la porción de la función `imprimir_pagina_debug` encargada de salvar la información de la pantalla la ventana de error se imprimía correctamente (dejando de lado el hecho de que no era posible restaurar la pantalla al estado anterior). No fue posible determinar la razón exacta del fallo ni corregirlo pero puede especularse que, tal vez, el hecho de haber pedido una página

libre para guardar la pantalla (a falta de una dirección específica, se optó por este mecanismo) haya influido. Se consideró que no afectaba de forma significativa la resolución del trabajo y es por eso que se entregó de esta forma.

Por lo demás, se concluye que se alcanzó el objetivo de desarrollar el sistema de acuerdo a las especificaciones indicadas.