La vida, como un comentario
de otra cosa que no alcanzamos,
y que está ahí al alcance
del *salto* que no damos.

Rayuela, Cortázar

DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico Nº 3

System Programming: `jmp far a lo desconocido`

Segundo cuatrimestre de 2018 — Organización del computador 2

Integrante	LU	Correo electrónico
Nicolás Hertzulis	811/15	nicohertzulis@gmail.com
Cynthia Liberman	443/15	cynthia.lib@gmail.com
María Belén Ticona	143/16	ticona.belu@gmail.com

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

El siguiente trabajo práctico consiste en desarrollar un sistema mínimo que permita correr como máximo 20 tareas concurrentemente, siendo todas de nivel usuario. El mismo simula la dinámica de un juego entre dos jugadores en un tablero donde se ubican sus correspondientes tareas. En sí, cada una de ellas puede leer información del tablero, moverse a una nueva posición o generar una copia idéntica.

En el tablero se dispone de un conjunto de frutas de distintos sabores donde cada sabor tiene asignado un puntaje determinado. Cuando una tarea se mueve a una posición donde hay una fruta, se come la fruta y aumenta el puntaje del jugador. En caso de haber tareas de ambos jugadores, se desata una batalla en la que solo quedan las tareas del jugador con mayor puntaje. El juego en sí consiste en mover las tareas a fin de alcanzar el mayor puntaje posible, finalizando cuando no queden frutas por comer o solo queden tareas de un solo jugador.

En el marco del modelo propuesto, el sistema desarrollado en el TP cumple el rol de arbitro del juego puesto que se encarga de capturar los problemas que pudiesen surgir. Por ejemplo, si alguna tarea comete una excepción la misma es eliminada del juego.

1.1. Objetivos

Los objetivos propuestos para el trabajo práctico consisten en la resolución de un conjunto de ejercicios donde se aplican de forma gradual los conceptos de *System Programming*, entre ellos se incluyen:

- Completar descriptores en la Tabla de Descriptores Globales (GDT), con dos niveles de privilegio, de usuario y Kernel. Para describir los segmentos de código, datos y video.
- Pasar a modo protegido.
- Completar las entradas en la Interruption Descriptor Table (IDT) para asociar las diferentes rutinas a todas las excepciones del procesador, las interrupciones de reloj y teclado, además de 3 entradas adicionales para interrupciones de software, para las Syscall `read`, `move` y `divide`.
- Escribir una rutina para limpieza de la pantalla, las rutinas asociadas a la interrupción del reloj y teclado, y las rutinas asociadas a las Syscall.
- Escribir rutinas necesarias para inicializar el directorio y tablas de páginas (PD y PT respectivamente). Para el área del Kernel el mapeo se hará usando identity mapping.
- Activar Paginación.
- Escribir rutinas que se ecarguen de la administración de la memoria en el área libre del Kernel y el área libre de tareas.
- Definir entradas en la GDT para ser utilizadas como descriptores de Task State Segment (TSS). Para todas las tareas, la inicial, la idle y las 20 tareas que pueden estar en juego de manera simultánea.
- Construír una función que inicialice las estructuras de datos del scheduler. Crear una función que devuelva el indice de la GDT de la próxima tarea a ejecutar. Realizar el intercambio de tareas por cada ciclo de reloj.

2. Desarrollo

2.1. Ejercicio 1

1. *Completar la Tabla de Descriptores Globales (GDT) con 4 segmentos, dos para código de nivel 0 y 3; y otros dos para datos de nivel 0 y 3. Estos segmentos deben direccionar los primeros 666MB de memoria. En la gdt, por restricción del trabajo práctico, las primeras 21 posiciones se consideran utilizadas y no deben utilizarse. El primer ındice que deben usar para declarar los segmentos, es el 22 (contando desde cero).*

En primer lugar, se declaró el array GDT de tamaño 50 para poder tener allí los descriptores pedidos en este item desde la posición 32, así como la entrada correspondiente para el segmento de video y los 20 descriptores de TSS (Task State Segment) necesarios más adelante (véase Ejercicio 6).

En este ejercicio se completaron los 4 segmentos pedidos de la siguiente manera:

- Base: 0x0 pues el mode
- Límite: 0x299FF
- Granularidad (G): 0x1
- Tipo: 0x0A en los segmentos de código (lectura, ejecución) y 0x02 en los segmentos de datos (lectura y escritura).
- Nivel de privilegios (DPL): 0x0 en los segmentos de datos y código que utiliza el kernel y 0x3 en los segmentos de código y datos que utilizan las tareas.
- Sistema (S): 0x1 pues no son segmentos de sistema sino de código y datos (0x0 significa que son de sistema.
- Presente (P): 0x1.
- L: 0x0 (se indica que se trabaja con IA-32).
- Tamaño (Sz): 0x1 (en 0 indica 16 bits, en 1 indica 32 bits en modo protegido).

Como el modelo de segmentación es de tipo *flat*, los 4 deben ocupar el total de memoria requerido (666MB) y tener el mismo tamaño. De allí que la base comience en 0x0.

Como el tamaño de la memoria expresado es de 698351616 bytes, en caso de expresar el límite en bytes su valor hubiese sido de (698351616-1) bytes. La representación de este número hubiera requerido de 32 bits cuando para el mismo se tenía solo 20 bits. Por esta razón fue necesario activar granularidad para poder expresar el límite en una unidad de mayor tamaño: páginas de 4 KB cada una.

De este modo, se necesitaron $666 \times 2^{10} KB/(4KB/Pag) = 666 \times 2^8 pág = 170496$. Por lo tanto el límite obtenido fue 170496-1 = 170495 = 0x299FF.

Por otro lado, para todas las entradas de la GDT se seteó el atributo L con 0x0 puesto que este valor indica que se trabajará con una arquitectura de IA-32. Su valor en 0x1 en cambio denota el uso de una arquitectura de tipo x86-64.

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

   Para ello realizamos los siguientes pasos:

   - Habilitar A20 para poder acceder a direcciones superiores a 2^{20} bits
   - Inicializar la GDT, cuyas entradas fueron ingresadas a partir de una dirección de memoria arbitraria, mediante la instrucción `lgdt`.
   - Setear en 1 el bit menos significativo de CR0 para activar modo protegido
   - Saltar a `selector_segmento_código_0:modoprotegido`. Para obtener el selector correspondiente se tomó el ındice del segmento de código de nivel 0 shifteado 3 bits a la izquierda dado que para los selectores se debe considerar: 1 bit de TI (LGT o GDT) y 2 bits del RPL. Con estos últimos 3 bits se eligió trabajar con la GDT y con RPL 0. En cuanto a `modoprotegido` corresponde a la dirección de memoria donde arranca el código que se ejecutará a continuación (ya en modo protegido).
   - Establecer los registros selectores de segmento datos: ds, es, gs y ss (índice en GDT de datos nivel 0 con TI y RPL en 0).
   - Establecer la dirección base de la pila del kernel en 0x27000.

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

   Como la pantalla es de 50x80 y cada celda ocupa 2 bytes, se necesitan direccionar 50*80*2-1 bytes = 7999 bytes. Luego, este número en hexadecimal es 0x1F3F. Dado que para escribir el mismo bastan los 16 bits del campo de límite, se mantiene la granularidad en cero.

   - Base: 0xB8000 pues lo indica el enunciado.
   - Límite: 0x1F3F
   - Granularidad: 0x0
   - Tipo: 0x02 (lectura, escritura).
   - Nivel de privilegios (DPL): 0x0 pues es de uso exclusivo del kernel.
   - Sistema (S): 0x1 (que significa desactivado) pues no es un segmento de sistema sino de video.
   - Presente (P): 0x1.
   - L: 0x0 (se indica que se trabaja con IA-32).
   - Tamaño (Sz): 0x1 (en 0 indica 16 bits, en 1 indica 32 bits en modo protegido).

4. *Escribir una rutina que se encargue de limpiar la pantalla y pintar 1 el área del tablero 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 vıdeo por medio del segmento de datos y no este último.*

   Se realizó un ciclo que recorre las celdas, pintando el fondo de negro y gris, borrando el texto. Se consideró que cada celda está compuesta por 2 bytes donde el byte menos significativo es el caracter a escribir y el más significativo es el atributo de color.

   - Byte de caracter: código ASCII
   - Byte de atributo: tiene formato BF donde B es un nibble (número de 4 bits) que indica el color de fondo (background color) y F es un nibble que indica el color del texto (foreground color).

2.2. Ejercicio 2

1. *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.*

2. *Hacer lo necesario para que el procesador utilice la IDT creada anteriormente. Generar una excepción para probarla.*

La IDT es un arreglo de descriptores de interrupción. Al producirse alguna se debe ejecutar una rutina que atienda la interrupción correspondiente. Cada una de las entrada de dicho arreglo cuenta con un selector de segmento de código donde está escrita la rutina, un offset para determinar donde inicia la misma y los atributos: presente, DPL (*Descriptor Previlege Level*) y Tipo.

Paraeste ejercicio se completaron las primeras 20 entradas de la IDT, siendo todas estas interrupciones internas (excepciones). Además, las siguientes 11 entradas están reservadas por diseño de arquitectura por lo que no se completarán. Las entradas restantes se encuentran disponibles para desarrollo.

- 0 a 19: Excepciones.
- 20 a 31: Reservados (no se deben usar).
- 32 a 255: Disponibles programador del sistema operativo.

En primer lugar se declararon las rutinas de atención a las interrupciones en el archivo `isr.h`, siendo estas `_isrN()` donde N es índice en el array IDT. Luego, se procedió definiendo los descriptores en la IDT en el archivo `idt.c` con el `#define IDT_ENTRY` otorgado por les docentes.

- offset: `&_isrN()` posición de memoria donde comienza la rutina de atención de la interrupción.
- segsel (Segment Selector): `0xC0` índice del segmento de código de nivel cero en la GDT shifteado 3 bits a la izquierda puesto que `TI` y `RPL` valen 0.
- attr (Attribute): `0x8E00` donde los ceros a la derecha están fijos por la arquitectura y el 0x8E corresponde con el siguiente formato:
  - P: `0b1` para indicar que es una entrada en uso
  - DPL: `0b00` dado que es una excepción del procesador
  - S (*Storage Segment*): `0b0` para indicar que no es fault (puerta de interrupción o trap).
  - Tipo: `0xE` para indicar que es una puerta de interrupción de 32 bits.

Cabe mencionar que el tipo de interrupción *Interrupt Gate* se caracterizan por deshabilitar otras interrupciones en caso de producirse alguna otra. De esta forma se evita interferir con la ejecución de la rutina de atención.

Una vez completados los descriptores, en `isr.asm` se definieron los mensajes a imprimir en pantalla indicando el tipo de excepción generada, imprimiéndose cada uno en pantalla en la rutina correspondiente -se empleó la macro `_isr%1`-.

2.3. Ejercicio 3

1. *Completar las entradas necesarias en la IDT para asociar una rutina a la interrupción del reloj y otra a la interrupción de teclado. Además crear tres entradas adicionales para interrupciones de software, 0x47, 0x49 y 0x4C.*
2. *Escribir la rutina asociada a la interrupción del reloj, para que por cada tick llame a la función nextClock. 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 nextClock está definida en isr.asm.*
3. *Escribir la rutina asociada a la interrupción de teclado de forma que si se presiona cual- quiera de 0 a 9, se presente la misma en la esquina superior derecha de la pantalla.*
4. *Escribir la rutina asociada a la interrupción 0x47 para que modifique el valor de eax por 0x42. Posteriormente este comportamiento va a ser modificado para atender uno de los servicios del sistema.*

Se procedió de manera análoga al ejercicio anterior a excepción de las interrupciones de software 0x47, 0x49 y 0x4C, cuyos atributos en el descriptor de interrupciones se completó de la siguiente forma:

- P: `0b1` para indicar que es una entrada en uso
- DPL: `0b11` dado que la syscall la deben poder llamar las tareas (nivel 3 de privilegio).
- S (*Storage Segment*): `0b0` para indicar que no es fault (puerta de interrupción o trap).
- Tipo: `0xE` para indicar que es una puerta de interrupción de 32 bits.

En cuanto al código en `isr.asm`, apenas se inicia cada una se preserva el estado inicial de todos los registros de propósito general, se restauran sus valores para volver al estado anterior de efectuarse la interrupción. Para esto último se tienen las instrucciones `pushad` y `popad` respectivamente.

Para las rutinas del reloj y del teclado, se debe al inicio del código llamar a la función `pic_finish1` para avisarle al PIC (*Programmable Interruption Controller*) que se ha atendido la interrupción efectuada y pueda así recibir otra (se llama a esta función dado que son dispositivos que emiten una señal a ser percibida por la CPU).

La rutina correspondiente al teclado obtiene el scan code entrante en el registro `al` leyendo el puerto número`0x60`. Este valor se pasa como parámetro a `print_digit`, función en C que se encarga de imprimir el caracter en la esquina de la pantalla en caso de ser un número.

2.4. Ejercicio 4

1. *Escribir las rutinas encargadas de inicializar el directorio y tablas de páginas para el kernel (mmu initKernelDir). 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 2.*

2. *Completar el código necesario para activar paginación.*

3. *Escribir una rutina que imprima el número de libreta de todos los integrantes del grupo en la pantalla.*

   En primer lugar, nótese que se pide mapear de `0x00000000` a `0x003FFFFF`, es decir, 4 MB para los cuales resulta necesario mapear 1024 páginas completas, lo cual es equivalente a mapear toda una tabla de paginación completa. Es por ello que para inicializar el directorio del kernel, se realizó una función `mmu_map_kernel_pages((PDE*) KERNEL_PAGE_DIR, (PTE*) KERNEL_PAGE_TABLE_0)` la cual toma la dirección del directorio (`0x27000`) y la página de la *page table* necesaria (página contigua a la del directorio, `0x28000`). Dicha función recorre todo el *page directory* seteando los siguientes atributos:

   - Present (P): 0b1 (presente)
   - Read/Write (RW): 0b1 (habilitada la escritura y lectura)
   - User/Supervisor (US): 0b0 (nivel supervisor)
   - BasePageTable: `KERNEL_PAGE_TABLE_0` $<<$ 12.
   - Resto de atributos en 0.

   Es importante resaltar que para indicar la base de las páginas como atributo tanto en un PTE como en PDE, se deben escribir los 20 bits más significativos de la dirección.

   Luego, se procede a inicilizar dicha *page table* con la función `init_new_pt(PTE* page_table, bool identidad, bool is_present, uint8_t u_s)`, la cual recorre toda la tabla seteando las 1024 páginas necesarias como presentes, de lectura/escritura y nivel *supervisor*. El parámetro `identidad` es un booleano que permite determinar los 20 bits de la dirección base de la página. Al tratarse de *identity mapping*, se escribe como base el valor del iterador en el ciclo que recorre las 1024 entradas.

   Una vez realizadas dichas funciones en el archivo `mmu.h` y `mmu.c`, se procedió a llamarlas desde `kernel.asm`, donde posteriormente se activó paginación mediante los siguientes pasos:

   - Se cargó en el registro de control `CR3` la `KERNEL_PAGE_DIRECTORY_ADDRESS`, `0x27000`.
   - Se seteó en 1 el bit menos significativo del registro de control `CR0` mediante un `OR` dado que no se quiere alterar los otros atributos de dicho registro.

   Por último, la impresión de las libretas se pudo efectuar con la macro brindada por la cátedra `print_text_pm`. La motivación se este ejercicio es la verificación de que la paginación efectuada haya sido realizada correctamente.

2.5. Ejercicio 5

1. *Escribir una rutina (mmu init) 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).*

2. *Escribir una rutina (mmu initTaskDir) encargada de inicializar un directorio de páginas y tablas de páginas para una tarea, respetando la figura 3. La rutina debe solicitar dos páginas del area libre de tareas 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 parámetros que considere necesarios.*

3. *Escribir dos rutinas encargadas de mapear y desmapear páginas de memoria.*

4. *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.*

En primer lugar, en la función `mmu_init()` se inicializaron las variables globales `mmu_next_free_kernel_page` y `mmu_next_free_task_page` con los valores 0x00100000 y 0x00400000 respectivamente. Además se dispuso de las funciones `mmu_get_next_free_kernel_page()` y `mmu_get_next_free_task_page()` que se encargan de actualizar dichas variables globales aumentando su valor en 4KB y retornando la dirección de una página disponible del área correspondiente.

Para administrar la memoria correspondiente a las tareas se decidió trabajar de manera estática para las 20 tareas posibles. Dicha decisión implicó tener una serie de estructuras donde se tengan las direcciones de los directorios, la tabla de páginas correspondientes al mapeo del área del kernel vía *identity mapping*, la tabla de páginas correspondientes a las páginas de la tarea en sí: código, pila de nivel 0 y pila de nivel 3.

- `directory_address[TASK_LIMIT]` con los directorios
- `table_address_kernel[TASK_LIMIT]` tablas para el área del kernel
- `table_address_task[TASK_LIMIT]` tablas para la tarea
- `code_address[TASK_LIMIT]` páginas para el código de tarea
- `user_stack_address[TASK_LIMIT]` páginas para la pila de nivel 3 de la tarea
- `kernel_stack_address[TASK_LIMIT]` páginas para la pila de nivel 0 de la tarea

Luego, se realizó la función `mmu_init_task_directory(uint8_t task_index)` la cual dado un índice correspondiente a una tarea se encarga de:

- Inicializar el directorio con la tabla de páginas correspondiente al identity mapping del área del kernel (mediante la función `mmu_map_kernel_pages((PDE*) (directory_address[task_index]), (PTE*) (table_address_kernel[task_index]))`)
- Mapear la pagina de código en la tabla de la tarea con la dirección virtual `0x08000000` con nivel *user*
- Mapear la pagina de pila de nivel 3 en la tabla de la tarea con la dirección virtual `0x08001000` con nivel *user*

No es necesario mapear aparte la página del directorio, ambas páginas de las tablas y la página de la pila de nivel 0 de cada tarea dado que las mismas pertenecen al área libre de kernel (el cual es mapeado por identity mapping en el primer item).

En cuanto a la función `mmu_map_page(uint32_t virtual, uint32_t cr3, uint32_t phy, PTE* new_table, uint8_t u_s)`, la misma se encarga de:

- Encontrar los índices correspondientes al directorio y la tabla de páginas en base a la dirección virtual pasada por paràmetro y realizar los shifteos necesarios.
- Hallar la base de la tabla de páginas si la entrada en el directorio tenía atributo presente o seteando el mismo con la dirección pasada por parámetro. En este último caso se inicializa toda la tabla con sus entradas como no presentes y se escriben los 20 bits más significativos en la entrada del directorio correspondiente al índice obtenido de la dirección virtual.
- Se setean los atributos de escritura/lectura, nivel de privilegio usuario y presentes en la entrada del directorio correspondiente a la tabla del item anterior
- En la entrada de la tabla de páginas correspondiente al índice obtenido de la dirección virtual, se desmapea cualquier dirección física de página previamente mapeada.

- Se llama a la función `set_new_page(base_table + pageTable_index, (phy 12), PAG_P, u_s)`, seteando dicha entrada en la tabla de páginas con atributo de presente, lectura/escritura y el nivel de privilegio pasado por parámetro.

  Es sumamente importante resaltar que la función de mapeo que se propone en este trabajo es específica para el contexto del TP, dado que se sabe que a la misma solo se la llamará para mapear las direcciones virtuales de código y pila de nivel 3 de las tareas. Es por ello que, por ejemplo, no se reciben como parámetros todos los atributos a setear, sin ir más lejos, siempre se mapea con lectura y escritura permitida. En el caso de otro contexto de TP en el cual alguna tarea solo debiese leer cierta parte de sus datos, se debiese pasar como parámetro el atributo `r_w` para que sea `read_only`. Por otro lado, la función recibe como parámetro un puntero de tabla de páginas puesto que se sabe que tras mapear el área del kernel en el directorio de la tarea, el mismo ocupó toda una tabla por completo (siendo necesario una nueva tabla para mapear más páginas). De allí que en lugar de pedir una nueva página del área libre de kernel, se pase la dirección correspondiente a cada tarea guardada en el array `table_address_task[TASK_LIMIT]` por el manejo estático de memoria optado. En esta misma tabla se mapean la páginas de código y pila de nivel 3.

2.6. Ejercicio 6

1. *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.*

2. *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.*

3. *Construir una función que complete una TSS libre con los datos correspondientes a una tarea. El código de las tareas se encuentra a partir de la dirección 0x00010000 ocupando una pagina de 8kb cada una según indica la figura 2. 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 initTaskDir. 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.*

4. *Completar la entrada de la GDT correspondiente a la tarea inicial.*

5. *Completar la entrada de la GDT correspondiente a la tarea Idle.*

6. *Escribir el código necesario para ejecutar la tarea Idle, es decir, saltar intercambiando las TSS, entre la tarea inicial y la tarea Idle.*

Para este ejercicio se realizó la función `tss_init_gdt_entry(uint32_t base, uint8_t index)` la cual inicializa una entrada en la GDT de acuerdo al índice pasado por parámetro y la dirección base de la TSS que describe la entrada en cuestión. Para todas las TSS los atributos definidos en la `GDT entry` son los siguientes:

- Límite: `0x67` dado que el tamaño mínimo para cada TSS es de 104 bytes, por lo que la mínima cantidad de bytes direccionables es 103.
- Base: `base` dirección pasada por parámetro a la función.
- Tipo: `0x9` valor correspondiente con atributos propios de un segmento tipo TSS, con el atributo de *busy* apagado.
- DPL: `0x0` puesto que las tareas no deben poder realizar un salto de tarea, es decir, no deben tener acceso a ningún segmento de TSS.
- Granularidad: `0b0` dado que el límite se mide en bytes.

- El resto de los atributos están en cero por características de TSS.

Una vez definidas las entradas en la GDT para la tarea inicial y la tarea *idle*, se procedió a completar la información de la TSS para esta última. Cabe resaltar que la funcionalidad de la tarea inicial es tener una tarea desde la cual realizar el primer cambio de contexto de ejecución saltando a la *idle*. La información que se guarda en la TSS y sus valores iniciales resultan superfluos, de allí que no se inicialice esta estructura.

En cuanto a la TSS de la tarea *idle*, siguiendo las pautas del enunciado se inicializa la estructura de la siguiente forma:

- `esp0`: `0x27000` se asigna la dirección numéricamente más alta de la pila del kernel.
- `stack_seg_0`: selector de segmento de datos nivel 0.
- `cr3`: `0x27000` dirección donde inicia la página del directorio del kernel.
- `EIP`: `0x14000` dirección donde inicia el código de la *idle*.
- `EFLAGS`: `0x00202` habilitadas las interrupciones (por ejemplo, la interrupción del reloj).
- `SS/EF/GS/DS/ES`: selector de segmento de datos nivel 0.
- `CS`: selector de segmento de código nivel 0.
- `iomap`: `0xFFFF` no se quiere que las tareas pueden emplear dispositivos de entrada/salida.
- Resto de atributos en 0.

Luego, se realizó la función `tss_init_user_task(tss* tss_user_task, uint8_t task_index)` que dado un índice correspondiente a un ID de una tarea, se encarga de inicializar la TSS con las direcciones de la pila de nivel 0, de nivel 3 y la página de código correspondientes reservadas para cada tarea en las estructuras de memoria estática (véase Ejercicio 6). De esta forma, la TSS para cada una queda inicializada de la siguiente manera:

- `esp0`: `kernel_stack_address[task_index] + PAGE_SIZE_4K` se asigna la dirección previamente guardada
- `stack_seg_0`: selector de segmento de datos nivel 0.
- `cr3`: `directory_address[task_index]`
- `EIP`: `0x08000000` dirección virtual donde inicia el código de la tarea.
- `EFLAGS`: `0x00202` habilitadas las interrupciones (por ejemplo, la interrupción del reloj).
- `SS/EF/GS/DS/ES`: selector de segmento de datos nivel 3.
- `CS`: selector de segmento de código nivel 3.
- `iomap`: `0xFFFF` no se quiere que las tareas pueden emplear dispositivos de entrada/salida.
- `esp/ebp`:`0x08002000` dirección virtual numéricamente más alta de la pila de nivel 3
- Resto de atributos en 0.

Una vez inicializadas todas la estructuras de las tareas, en `kernel.asm` se cargó la tss de la tarea inicial mediante la instrucción `ltr SEL_TSS_INICIAL` la cual se encarga de cargar en el *Task Register* (registro que contiene el selector de la tarea que se encuentra en ejecución) la TSS de la tarea inicial. A continuación para efectuar el intercambio de tarea se realiza un `jmp far` puesto que se realiza un cambio de contexto de ejecución: `jmp far selector_tarea_a_saltar:0`. De esta forma se saltó de la tarea inicial a la idle.

2.7. Ejercicio 7

1. *Construir una función para inicializar las estructuras de datos del scheduler.*
2. *Crear la función sched nextTask que devuelve el ındice 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.*
3. *Modificar las rutinas de la interrupciones 0x47, 0x49 y 0x4C para que implementen los distintos servicios del sistema según se indica en la sección 3.1.*
4. *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 nextTask().*
5. *Modificar las rutinas de excepciones del procesador para que desalojen a la tarea que estaba corriendo y ejecuten la próxima.*
6. *Implementar el mecanismo de debugging explicado en la sección 3.3 que indicará en pantalla la razón del desalojo de una tarea.*

El scheduler fue diseñado como un componente aislado del sistema, especialmente aislado de la lógica de juego. Soporta 20 tareas de usuario. Tiene internamente una variable que guarda el índice en la `GDT` de la tarea que se está ejecutando y un arreglo de booleanos de 20 posiciones que indica cuáles tareas se encuentran programadas para ejecutarse.

Para el cálculo de la próxima tarea la función `sched_next_task` recorre el arreglo de booleanos desde el índice actual hasta encontrar la próxima tarea programada. Si llegó al final devuelve el selector de la tarea *idle* para indicar que ya se ejecutaron todas las tareas de usuario y en la siguiente llamada arranca desde el comienzo. Si la tarea actual es *idle* y no hay tareas programadas, devuelve el selector de la tarea nula para indicar que no hay próxima tarea.

Consideraciones especiales en la implementación de las syscall:

- Cada tarea tiene un campo `read_count` que cuenta la cantidad de llamados a `read` en el turno actual y se resetea al fin de cada turno.
- Se agregó el campo `ReadLimitExceeded` al enum `datatype` para devolver en caso que se haya superado el límite de lecturas.
- `read`: en la misma posición puede haber fruta y una o más tareas amigas y enemigas. En ese caso es decisión nuestra qué devolver, ya que no se puede dar toda la info. Se decidió devolver siempre fruta en caso de que haya. Si no hay fruta y hay jugadores de ambos bandos, se devuelve siempre oponente. La lógica de esto es la siguiente: si devuelve que hay un amigo, sabemos que no hay oponentes y no es interesante moverse a la posición de un amigo porque él se come la fruta (a menos que queramos armar un ejército para pelear en cuyo caso también podríamos hacerlo); en cambio, si devuelve oponente, podemos decidir si meternos a pelear o escaparnos (no tenemos forma de saber si hay más de un oponente o si incluso hay uno o más amigos para apoyarnos).
- En `move` se transforma la distancia y la dirección indicadas en dos desplazamientos, uno en la coordenada `x` y otro en la coordenada `y`. En caso de que alguna coordenada resultante luego del desplazamiento fuera mayor a cincuenta o menor a cero se le suma 50 para que aparezca en el borde opuesto.
- Al llamar a `divide` con peso mayor a 1, la tarea pierde la mitad de su peso automáticamente, haciendo que si al fin de turno hay una pelea, ésta utilice la mitad de su peso para pelear. La nueva tarea creada por divide aparece recién en el siguiente turno, por lo cual no entraría en la eventual pelea a menos que los oponentes ganen y en el siguiente turno no se muevan. En ese caso habría una segunda pelea al fin del siguiente turno.

- `divide` se encarga de dejar la tarea lista para ser ejecutada: escribe el código de la tarea en la dirección física correspondiente, inicializa la tss como una nueva tarea y copia los metadatos de la otra tarea (peso, posición, etc.). Esta nueva tarea va a retomar desde la interrupción `divide`, para lo cual se hacen los siguientes pasos:
  - Copiar la pila de nivel cero que contiene los valores de los registros salvados de la tarea original y la dirección de retorno para `iret`.
  - Sumarle al `esp` de la nueva `tss` el offset en la pila de nivel cero de la tarea original.
  - Setear en los registros de segmento de código y pila (`cs` y `ss` respectivamente) los selectores de segmento de nivel cero (y no los de nivel 3 como una tarea que inicia sola).
  - Setear en `eip` la dirección de una función muy sencilla cuyo código se encuentra en el área de kernel que simplemente hace los `pop` correspondientes, setea el resultado en `eax` y hace iret para continuar la ejecución en el cuerpo de la tarea de nivel 3.
- Las tres syscall utilizan una variable local en la pila para devolver el resultado, ya que de otra manera el valor resultado que se encuentra en `eax` sería pisado por `popad`.
- Antes de saltar a la tarea `idle`, las syscall informan de esto al scheduler para no quebrantar su mecanismo.
- Cada una de las syscall tiene su correspondiente entrada en la IDT con los siguientes atributos:
  - Tipo de puerta: puerta de interrupción (0xE)
  - Presente: sí (0x1)
  - Nivel de privilegios: usuario (0x3)
  - Segmento de almacenamiento (S): 0 porque es una interrupción

El intercambio de tareas se realiza en la interrupción de reloj. En primer lugar se llama a la función `sched_next_task`. Si la función devuelve el selector de la tarea nula o el selector de la tarea idle se considera que es fin de turno y se llama a la función `game_next_step`. Si había devuelto el selector de la tarea nula, se considera que la interrupción de reloj actual interrumpió a la tarea idle y no hay más tareas programadas, por lo cual se llama directamente a `iret` sin hacer `jmp far` para volver a idle hasta la próxima interrupción de reloj. En cualquier otro caso (ya sea que había devuelto el selector de idle o de una tarea de usuario) se hace el `jmp far` a la nueva tarea.

La función `game_next_step` realiza las siguientes acciones:

1) Se actualizan las posiciones de las tareas ”vivas”, la posición actual, pasa a ser la que tendrá en el próximo turno. Esta es distinta a la actual si en el turno recién transcurrido, se llamó a la Syscall `move`.

2) Se crea una nueva estructura *equal_positions*, la misma es una matriz de 20x20, que contendrá un puntero a cada una de las tareas (donde el struct tarea apuntado contiene toda la info de estado de la tarea), mientras que los espacios libres de la matriz apuntaran a Null. En cada columna se ingresan los punteros a la tarea que se encuentren en la misma posición del tablero. De este modo si todas las tareas se encuentran en distintas posiciones del tablero, la primer fila se encuentra completa, mientras el resto de la matriz está libre. La contrapartida de este caso es cuando todas las tareas se encuentran en la misma posición tanto las del jugador A como las de B, pudiendo haber un máximo total de 20 tareas, estas se ubican todas en la primer columna de la matriz mientras el resto de las columnas quedan vacias (registrese que luego del paso (1) las posiciones son las que tendrán las tareas en el próximo turno y por lo tanto aún están vivas las tareas de ambos jugadores).

3) Se da vida a las nuevas tareas que se crearon por la Syscall `Divide` en el turno recién finalizado. Esto implica pasar su estado de FUTURE_TASK a TO_EXECUTE.

4) Se mata a las tareas, aquellas tareas que por la razón que fuere (excepto por guerra de tareas) finalizaron su ejecución en este turno. Esto implica pasar su estado de NEXT_FREE a FREE.

5) Se llama a la función *guerra_de_tareas*.

6) Se llama a la función *actualizar_pantalla*.

En la función *guerra_de_tareas*, se recorre la matriz equal_positions columna por columna y se almacena en cada posición de un arreglo de tamaño 20, la suma de pesos de las tareas de A y las tareas de B por separado, en el indice correspondiente a cada columna. Con esta información se podrá decidir cual es el ganador en cada casillero ocupado y matar a las tareas perdedoras pasando su estado a FREE. Si hubiere frutas en el casillero el ganador suma a su puntaje el valor de las frutas, y la fruta pasa su estado a NONE.

Para que todas las excepciones desalojen a la tarea en ejecución, se realizó la siguiente funcionalidad en la macro de excepciones:

- Pedir al *scheduler* el ID de la tarea actual.
- Llamar con este ID a `kill_monster`, la cual setea el estado en `NEXT_FREE` para que al final de turno, luego de calcular el resultado, esta pase a `FREE` y sea removida del *scheduler*.
- Llamar a una función que se encarga de pintar el cartel del modo debug y setear una variable global de assembly en TRUE indicando que se está en modo debug. Esta variable es leída por la interrupción de reloj y en caso de ser TRUE, se evita llamar a `sched_next_task` y `game_next_step` hasta que el usuario presione la tecla jota y desactive el modo debug.

3. Conclusiones

En primer lugar, habiendo seguido el orden de la propuesta de ejercicios para desarrollar un sistema que permita correr 20 tareas concurrentemente, se logró concretar un sistema que cumple con las pautas dadas en el enunciado.

Sin ir más lejos, se logró realizar un pasaje de modo real a modo protegido, elaborar una segmentación de tipo *flat*, realizar un esquema de paginación para el kernel y las tareas ajustándose a los requerimientos pedidos, configurar el manejo de tareas e interrupciones desarrollando las estructuras y lógicas correspondientes.

Cabe resaltar que durante el desarrollo del sistema, se decidió optar por un manejo de memoria estática, preservando al iniciar la simulación del juego las direcciones de las páginas de memoria necesarias para el máximo de tareas posible. De esta forma, se evitó un abuso del uso de memoria en comparación con la lógica de pedir una nueva página (ya sea del área libre de kernel o del área libre de tareas) cada vez que fuese necesario crear una tarea nueva.

Finalmente, es importante remarcar que la funcionalidad de los métodos involucrados en la paginación son específicos para el contexto de este trabajo práctico. Por practicidad se omitió el manejo de casos posibles de mapeo de direcciones que no se aplicasen en este juego, siendo un manejo no genérico.