DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico III

## System Programming - Low Power

Organización del Computador II

Segundo Cuatrimestre de 2020


| Integrante | LU | Correo electrónico |
|---|---|---|
| Marco Sotomayor | 731/14 | marco.soto1995@gmail.com |
| Walter Tejera | 362/15 | wtejerac@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. 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. Los ejercicios están inspirados en la serie Rick and Morty.

El trabajo busca construir un sistema mínimo que permita correr como máximo 22 tareas concurrentemente a nivel de usuario. Este sistema simulará un universo en el que se encuentran 2 jugadores, Rick y Morty, y un tablero donde se moveran las tareas (mr meeseeks) buscando sumar puntos al recolectar objetos(mega semillas) repartidos por el mapa. Los meeseeks serán creados a partir de la acción de los jugadores y se moverán sobre el tablero de forma autónoma. Cuando un meeseeks llega a alguna de las semillas tanto el meeseeks como la semilla serán eliminadas del mapa y se sumaran 425 puntos al invocador del meeseeks.

Los ejercicios de este trabajo práctico proponen utilizar los mecanismos que posee el procesador para la programación desde el punto de vista del sistema operativo.

# 2. Introducción

Para este trabajo se utilizará como entorno de pruebas el programa Bochs. El mismo permite simular una computadora IBM-PC compatible desde el inicio, y realizar además tareas de debugging. Todo el código provisto para la realización del presente trabajo está ideado para correr en Bochs de forma sencilla.

Al iniciar, una computadora 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.

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.

# 3. Desarrrollo

## 3.1. Ejercicios

### 3.1.1. Ejercicio 1

a) 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 201MB de memoria. En la gdt , por restricción del trabajo práctico, las primeras 10 posiciones se consideran utilizadas y por ende no deben utilizarlas. El primer índice que deben usar para declarar los segmentos, es el 10 (contando desde cero).

Editamos el archivo `gdt.c`, colocando dentro del arreglo de entradas de gdt definido en el archivo inicial, agregando cuatro nuevas entradas, una correspondiente a cada segmento pedido. Las posiciones usadas del arreglo son las primeras 4 despúes de la entrada 9 ya que estas se encuentran reservadas. Para que sea mas fácil reconocer cada segmento cada posición tiene un nombre, el cual previamente definimos en el archivo `defines.h`, con el valor númerico que corresponde a su posición.

Cada entrada de la GDT consiste en un struct `gdt_entry`, definido en `gdt.h`, cuyos valores varían según las características del segmento al que apunta. Todos los segmentos definidos tienen base en la posición `0x00000000` y límite en `0xC8FF` con granularidad 1 para cubrir los primeros 201MB de memoria. Dependiendo de si es un segmento de nivel 3 o 0, el *DPL* del descriptor de segmento cambiara por `11` o `00` respectivamente.

b) Completar el código necesario para pasar a modo protegido y setear la pila del kernel en la dirección `0x25000` (es decir, en la base de la pila).

Modificamos el archivo `kernel.asm` agregando una línea para cargar la dirección de la GDT, con la instrucción `LGDT`, usando la variable `GDT_DESC definida` en `gdt.h`. Luego habilitamos la A20 Gate para setear el bit de PE (Protected Environment) del registro CR0 en 1 y finalmente realizamos un *jump far* al segmento de código de nivel 0 definido en el punto anterior. El offset esta dado por una etiqueta que agregamos abajo, llamada `modoprotegido`. Al realizar este salto tambien debimos de agregar una línea que indique el paso a 32 bits.

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

Editamos el archivo `gdt.c` para agregar una nueva entrada en la GDT, siguiente a las anteriores y apuntando a un segmento con base `0xB8000` y límite `0x1F3F`. Como el tamaño de la pantalla es de 80x50, la cantidad de píxeles totales es de 4000 px. Sumado a esto, cada pixel es representado por 2 bytes, dando así 8000 bytes para la pantalla por lo que la última dirección accesible es `0xB8000 + 0x1F3F` (7999 en decimal).

d) Escribir una rutina que se encargue de limpiar la pantalla y pintar el área del mapa con algún color de fondo, junto con las barras de los jugadores según indica la sección 0.4.5. 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 video por medio del segmento de datos.

El ejercicio esta diseñado para que podamos ver como escribir en memoria utilizando un segmento distinto al segmento de datos que tenemos seteado por defecto. Para pintar la pantalla iteramos desde `mov [fs:reg]` donde `fs` es el selector de segmento de video, con TI en 0 y RPL 00, y `reg` un registro cualquiera. Una vez que `reg` vale `0xA0` significa que llegamos al final de la primera fila por lo que empieza el mapa. La terminación del mapa se encuentra en el byte `0x19A0`. Luego solo queda pintar de negro nuevamente el resto de la pantalla, hasta el byte `0x1F3F`.

### 3.1.2. 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.

En el archivo `idt.c` modificamos la función `idt_init` para inicializar los 31 (0-30) tipo de excepciones de sistema. Cada entrada tiene seteados los siguientes atributos:

- `off_set`: La dirección en la cual empieza la rutina de la excepción correspondiente
- `select_seg`: El selector de segmento con el que se correrá la rutina. Tiene que apuntar a un segmento de código.
- `attr`: bits correspondientes a los atributos de una `IDT_ENTRY`, como el DPL, el bit de Presente, el tamaño de la compuerta, entre otras.

  Estas interrupciones de excepciones tienen un Selector de segmento apuntando a un segmento de código de nivel 0 con un RPL en 00 y un TI en 0. con el bit de Presente en 1 y el tamaño de la compuerta de 32 bits.

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

En el archivo `kernel.asm` inicializamos la IDT con la función `idt_init`, para luego setear el puntero a la IDT con LIDT con la variable `IDT_DESC`, declarada en `idt.h`.

### 3.1.3. Ejercicio 3

a) 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 cuatro entradas adicionales para las interrupciones de software 88, 89, 100, 123.

Agregamos 3 entradas más (la 32, 33 y 47) asociadas a la interrupción de reloj, teclado y software respectivamente.

Para la interrupción de teclado y de reloj, tendremos un selector de segmento con RPL 00 y TI en 0. Sus atributos son: el bit de presente en 1, el DPL del segmento en 00 y el tamaño de la compuerta de 32 bits.

Para la interrupción de software (syscall) tenemos un selector de segmento de código de nivel 0 con RPL 00 y TI en 0. El bit de Presente en 1, el DPL del segmento en 11 (para que las tareas con privilegio de usuario, puedan llamar a esta syscall) y un tamaño de compuerta de 32 bits.

b) Escribir la rutina asociada a la interrupción del reloj, para que por cada *tick* llame a la función `nextClock`. La misma se encargara 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`.

En el archivo `isr.asm` agregamos la interrupción de reloj, que corresponde a la entrada 32 de la IDT , y consiste en guardar los registros, llamar a la función `pic_finish1`, indicando que ya llegamos a atender la interrupción para llamar luego a `nextClock`, recuperar los registros y volver de la interrupción con un `iret`.

c) Escribir la rutina asociada a la interrupción de teclado de forma que si se presiona cualquiera de 0 a 9, se presente la misma en la esquina superior derecha de la pantalla.

Modificamos el archivo `isr.asm` para agregar la interrupción de teclado, que corresponde a la entrada 33 de la IDT , y que consiste en llamar a `pic_finish1`, guardar la información de los registros, leer el código ingresado por teclado mediante in 0x60, verificar que no sea mayor a 0x80 (es decir, solté la tecla) y llamar a `print_hex` creada en el archivo screen.c que de acuerdo al código recibido, imprime la tecla correspondiente en la pantalla . Finalmente recupera la información de los registros y vuelve de la interrupción.

d) Escribir las rutinas asociadas a las interrupciones 88. 89, 100 y 123 para que modifique el valor de eax por 0x58, 0x59, 0x69, y 0x7b, respectivamente. Posteriormente este comportamiento va ser modificado para atender cada uno de los servicios del sistema.

Estas funciones fueron implementadas en `isr.asm` y modifican el valor de `eax` por los valores pedidos, para luego volver de la interrupción.

### 3.1.4. Ejercicio 4

a) 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 0x25000 y las tablas de páginas según muestra la figura 2.

Creamos dos structs: `tr_page_directory_entry` y `str_page_table_entry` que mediante metodos modelan un cada campo de un directorio de paginas y una tabla de paginas respectivamente. Modificamos la función `mmu_initKernelDir` del archivo `mmu.c`, definiendo la primer entrada del Directorio de Páginas en la dirección `0x00025000`, seteando los bits de *presente*, *read/write* en 1 y la base del la Tabla de páginas en `0x00026`, mientras que las demás entradas, las inicializamos en 0.

Luego definimos cada entrada de la Tabla de Páginas situada en la dirección anteriormente mencionada seteando los bit de *presente* y *read/write* en 1 y definiendo la base usando identity mapping(cada base corresponde al indice en la tabla).

Tanto la entrada del Directorio de Páginas como todas las entradas de su Tabla de Páginas correspondientes tiene privilegios de supervisor (el bit de *U/S* esta en 0). También cabe aclarar la entrada del Directorio de Páginas que seteamos tiene el bit de *PS* (Page Size) en 0, lo que significa que las páginas tienen tamaño 4Kb. Con esto logramos tener un direccionamiento de 4Mb de memoria con *identity mapping*.

b) Completar el código necesario para activar paginación.

Modificamos el archivo `kernel.asm` agregando líneas para modificar el registro `cr3` (que contiene la base del directorio de la tarea actual, en este caso el kernel), para que este apunte a la dirección `0x00025000` y tenga activo el bit de paginación de cr0.

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

La función `imprimir_libretas` se encuentra en `screen.h` y se encarga de imprimir un bloque negro en el centro de la pantalla con las libretas de los integrantes.

### 3.1.5. Ejercicio 5

a) Escribir una rutina (`mmu_init`) que se encargue de inicializar las estructuras necesarias para administrar la memoria en el área libre de kernel.

La función solo se encarga de inicializar la variable `next_free_kernel_page`, y se encuentra en el archivo `mmu.c`.

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

I- `mmu_mapPage(uint32_t virtual, uint32_t cr3, uint32_t phy)`.

Permite mapear la página física correspondiente a phy en la dirección virtual virtual utilizando cr3.

Agregamos dos parametros a la función: `uint8_t rw` y `uint8_t us`. Esta función, implementada en el archivo `mmu.c`, toma una dirección de memoria virtual, una de memoria física, un cr3, rw y us.

Primero revisa que esté el bit de presente en 1 en la entrada que marca la dirección virtual del Page Directory correspondiente al cr3, y en caso de no estarlo, la inicializa, utilizando una nueva página libre del area libre del kernel para crear su correspondiente Page Table.

Luego busca la entrada del Page Table y la inicializa, de modo que mapea a la dirección de memoria física pasada por argumento. Siempre seteando los bits de privilegios y permisos de lectura y escritura de manera acorde a los valores pasados por parámetro.

II- `mmu_unmapPage(uint32_t cr3, uint32_t virtual)`

Borra el mapeo creado en la dirección virtual virtual utilizando cr3.

Esta otra función, también implementada en el archivo `mmu.c`, simplemente limpia el bit de presente de la entrada del Page Table, de la correspondiente entrada del Page Directory del cr3 pasado por argumento.

c) Escribir una rutina (`mmu_init_task_dir`) encargada de inicializar un directorio de páginas y tablas de páginas para una tarea, respetando la figura 3. La rutina debe mapear 4 páginas virtuales para la tarea Rick o Morty, a partir de la dirección virtual 0x1D00000. Esta función debe encargarse de copiar el codigo de la tarea desde la memoria del kernel a la pagina física correspondiente(0x1D00000 para Rick y 0x1D04000 para Morty). Sugerencia agregar a esta función todos los parámetros que considere necesarios.

Agregamos la función `mmu_init_task_dir` en el archivo `mmu.c`, que toma los siguientes parametros: `paddr_t phy_start, paddr_t code_start, size_t pages, vaddr_t v_start, uint8_t rw, uint8_t user_supervisor`. Define un nuevo Directorio de Páginas (pidientdo una nueva página del área libre del kernel), realiza identity mapping en la parte de la memoria destinada al kernel y al área libre del kernel. Se mapea `v_start` a `phy_start` tanto el cr3 actual como en el nuevo. Utilizando

el cr3 actual se escribe el codigo de la tarea (`code_start`) a travez de `v_start`. Luego se desmapea `v_start` del cr3 actual. La función devuelve el nuevo cr3.

d) A modo de prueba, construir un mapa de memoria para tareas e intercambiarlo con el del kernel, luego cambiar el color de fondo de la pantalla y volver a la normalidad. Este item no debe estar implementado en la solución final.

Para este ejercicio creamos

### 3.1.6. Ejercicio 6

a) Definir las entradas en la GDT que considere necesarias para ser usadas como descriptores de TSS. Mínimamente, una para ser utilizada por la tarea inicial y otra para la tarea idle.

En en archivo `gdt.c` agregamos 24 nuevas entradas de la GDT. Las primeras dos son para la tarea inicial y la tarea idle, las siguientes dos son para Rick y Morty y las ultimas 20 son las tss correspondientes a los Mr. Meeseeks de cada jugador.

Estas entradas tendrán un offset límite de 103 bytes y el bit de granularidad en 0, haciendo que se puedan acceder a 104 bytes del segmento de TSS. Todos tienen un DPL en 00 ya que solo quiero que el Kernel tenga acceso a estos segmentos. Las tss de las tareas inicial, idle, Rick y Morty tendran el bit de presente en 1, mientras que las de los Mr. Meeseeks permaneceran en 0 hasta que sean creadas por algun jugador.

b) Completar la entrada de la `TSS` de la tarea `Idle` con la información de la tarea `Idle`. Esta información se encuentra en el archivo `tss.c`. La tarea `Idle` se encuentra en la dirección 0x00018000. La pila se alojará en la misma dirección que la pila del kernel y será mapeada con identity mapping. Esta tarea ocupa 1 página de 4KB y debe ser mapeada con identity mapping. Además, la misma debe compartir el mismo CR3 que el kernel.

Definimos en `tss.c` la variable `tss_iddle` que toma el cr3 actual mediante la funcion `rcr3()`, con la misma pila que nuestro kernel (0x25000), el eip en 0x18000. Luego, le seteamos los selectores de segmento de la siguiente forma:

- `es`: GDT_IDX_DATA_0 $\ll 3$
- `cs`: GDT_IDX_CODE_0 $\ll 3$
- `ss`: GDT_IDX_DATA_0 $\ll 3$
- `ds`: GDT_IDX_DATA_0 $\ll 3$
- `fs`: GDT_IDX_DATA_0 $\ll 3$
- `gs`: GDT_IDX_DATA_0 $\ll 3$

Tanto GDT_IDX_DATA_0 como GDT_IDX_CODE_0 son reemplazos sintacticos definidos en defines.h con los indices al segmento de datos y codigo de nivel 0 respectivamente.

c) Completar la entrada de la GDT correspondiente a la tarea *inicial*.

d) Completar la entrada de la GDT correspondiente a la tarea *idle*.

En estos dos ejercicios, usamos la funcion `tss_init()` la cual setea las bases de las tss en la gdt, ya que no pueden resolverse en tiempo de compilación.

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

En `kernel.asm`, realizamos un `jmp far` a la entrada de la GDT donde se encuentra ubicado el descriptor de la TSS de la tarea Idle con offset 0.

f) Construir una función que complete una TSS con los datos correspondientes a una tarea. Esta función será utilizada más adelante para crear una tarea. El código de las tareas se encuentra a partir de la dirección 0x00010000 ocupando cuatro páginas de 4kb 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 del código de la tarea. Para el mapa de memoria se debe construir uno nuevo utilizando la función `mmu_init_task_dir`. Además, tener en cuenta que cada tarea utilizará una pila distinta de nivel 0, para esto se debe pedir una nueva página del área libre de kernel a tal fín.

Para este ejercicio, tendremos un array de tss, llamado `tss_tasks` de 22 posiciones, correspondientes a Rick, Morty y a los Mr Meeseeks de cada uno. Rick y morty estaran en `tss_tasks[0]` y `tss_tasks[1]` mientras que los Mr. Meeseeks estarán en las posiciones restantes. En las posiciones pares, tendremos a los Mr. Meeseeks de Rick y en las impares los de Morty.

Debido a esto separamos la creación de la tss de una tarea en dos:

- creación de tss para Rick y Morty: Definimos en la funcion `tss_init` los valores tanto para rick como para Morty. Estos se crearan con un nuevo cr3 (con el codigo de las tareas mapeado en la direccion virtual 0x1D00000), con la pila en la direccion virtual 0x1D04000 y los selectores de segmento con RPL 3, salvo el ss0 que necesitamos que sea un segmento de datos de nivel 0.

  El con esp empezará en la base de la nueva pagina libre que nos devuelve `mmu_next_free_kernel_page()`
- creación de tss para un Mr. Meeseeks: Se explayara mas sobre esto mas adelante.

### 3.1.7. Ejercicio 7

a) Construir una función para inicializar las estructuras de datos del scheduler.

En `sched_init` se inicializan variables globales que controlan los puntajes, la cantidad de meeseeks activos, la tarea actual que esta corriendo y 2 arrays uno de semillas y el otro de meeseeks. Cada elemento de los array contiene una instancia de semilla o meeseek respectivamente. En el caso de meeseeks, tenemos en cuenta sus coordenadas, si uso o no la portal gun, distancia actual de movimiento por turno y la cantidad de ticks de reloj dedicados a ese meeseek. Una semilla se compone de sus coordenadas en el mapa y si fue asimilada o no. Por último esta función se encarga de imprimir las semillas en pantalla.

b) Crear la función `sched_next_task()` que devuelve el indice en la GDT de la próxima tarea a ser ejecutada. Construir la rutina de forma que devuelva una tarea por cada jugador por vez según se explica en la sección 0.4.3.

La función fue implementada en `sched.c` y se llama desde la interrupción de reloj. Devuelve el selector de segmento de la siguiente tarea, en este caso Rick o Morty.

c) Modificar el código necesario para que se realice el intercambio de tareas por cada ciclo de reloj. Cargar las tareas Rick y Morty y verificar que se ejecuten (escriben su reloj cuando están activas, ejecutar system calls de prueba). El intercambio se realizara según indique la función `sched_next_task()`.

En `kernel.asm` cargamos la tss de la `tarea inicial` con `ltr` y a continuación saltamos a la tarea `Idle`. Una vez termine el quantum llegará la interrupción de reloj desde donde se llamara a `sched_next_task()`. El resultado devuelto es comparado para saber si la proxima tarea a saltar es la misma que estoy corriendo actualmente. De ser asi, obviamos el intermcabio de tareas, ya que se produciria un error porque el descriptor de tss a la que apunta el resultado de `sched_next_task()` tiene el bit de busy en 1. De ser distintos, pasamos a hacer el jump far al descriptor de tss que nos devolvio esta función.

d) Modificar las rutinas de interrupciones 88, 89, 100 y 123, para que pasen a la tarea idle cuando son invocadas. Verificar que la tarea idle se ejecuta (escribe su reloj).

En cada interrupción pedida realizamos un jump far al selector de segmento de la tss idle con RPL 0.

e) Modificar las rutinas de excepciones del procesador para que desalojen a la tarea que estaba corriendo y ejecuten la próxima. En caso de que se desaloje una tarea principal, el juego finaliza.

Modificamos las rutinas de excepciones para que en el caso de que uno de los 2 jugadores cometa la excepcion, se imprime un mensaje informando al ganador (el jugador contrario al que realizo la excepción) y no se realizan mas acciones en el campo.

f) Implementar el mecanismo de debugging explicado en la sección 0.4.4 que indicara en pantalla la razón del desalojo de una tarea.

Para este punto, tendremos dos variables más llamadas `modo_debug` y `modo_debug_corriendo`. El primero se usara para tener un control de si esta activado o no el modo debug, mientras que el otro se usara para saber si esta actualmete mostrandose la pantalla del modo debug. Usaremos la tecla `Y` para activar el modo debug mediante la interrupción de teclado en el archivo `isr.asm`. Esta rutina, verificara si se apreto dicha tecla y de ser así, se llamara a la fución `cambiar_modo_debug()`.

Esta función encendera el modo debug si esta apagado y de estar encedido, verificara si se esta mostrando la pantalla de debug para apagarla, más no desactiva el modo debug.

Por otra parte, en las rutinas de excepciones, antes de ejecutar algún veredicto sobre la tarea que cometió la excepción, verificaremos que el modo debug este encendido. De ser así, pushearemos todos los registros importantes para poder imprimir los valores pedidos mediante la función `mostrar_pantalla_debug`. Previamente a esto, activamos la interrupciones nuevamente con `sti`. Esto ultimo es necesario, para que cuando estemos la funcion previamente mencionada pueda se interrumpida por el teclado el cual como dijimos se encargara de cambiar el estado del modo debug.

Una pequeña cosa a considerar es que al activar las interrupciones, también nos podra interrupir el reloj, por lo que en dicha rutina, verificaremos que no estemos mostrando la pantalla del modo debug, antes de proseguir con las demas instrucciones.

### 3.1.8. Ejercicio 8

1. Inicializar la pantalla del juego, distribuyendo Mega Semillas por el mapa en posiciones aleatorias.

   Esto lo implementamos en la función `sched_init()`. Utilizamos la función rand() provista por la cátedra para generar números aleatorios, a los cuales les tomamos modulo `80` y `40` respectivamente para posicionar cada semilla en el mapa. Luego, utilizamos las funciones de `screen.c` para dibujar estas mega semillas en la pantalla.

2. Implementar la lógica de la rutina de interrupciones `meeseeks (88)` para crear un nuevo Meeseek. Verificar que el mapeo se realice en las direcciones correctas, que funcione la lógica de asimilación de **Mega Semillas** y que este esté agregado al scheduler. Actualizar el mapa.

   Para este ejercicio modificamos nuestra función `sched_next_task()` para agregar la lógica de la conmutación de tareas que contempla a los Mr. Meeseeks vivos. Ahora nuestra variable `tarea_actual` sera un numero de 0 a 21. Cuando ya haya pasado el turno tanto de Rick como de Morty, la función buscará al primer meeseek vivo que encuentre para devolverlo.

   En la rutina de interrupción se checkean varias cosas:

   - Que la función solo sea llamada por Rick o Morty. Caso contrario, se mata a la tarea.
   - Que los valores pasados por parametro (`code_start`, `x` e `y`) sean validos, es decir, que en la dirección desde donde debemos copiar el código del Meeseek este mapeado en la tarea y que tanto x como y no se pasen de 80 ni 40 respectivamente.
   - Que todavía haya espacio para crear un nuevo Mr. Meeseek.

   Una vez pasadas estas validaciones, pasamos a crear el mr meeseek con la función `crear_meeseek`. Esta función checkea que en las coordenadas creadas no haya una semilla. De haberla, asimilamos

la semilla poniendo su flag de encontrada en 1 y sumando 425 puntos al jugador correspondiente para finalmente obviar la creación del meeseek. De no haber una mega semilla en las coordenadas dadas, buscaremos en nuestro array de meeseeks (recorriendo elementos pares para Rick e impares para Morty) la primera posición en la que no exista un meeseek. para crearlo con las coordenadas pasadas.

Finalmente incializaremos la tss del meeseek con la función `tss_task_init`. Esta función inicializa el cr3 de la tarea (asignandole el cr3 del jugador creador), copiando los 2kb de código desde la dirección dada por `code_start` y mapeandola en alguna dirección virtual empezando desde 0x8000000 según corresponda. Luego, setearemos el esp0 en la base de la dirección que nos de la función `mmu_next_free_kernel_page`. Tomamos en cuenta la sugerencia dada por la catedra para reutilizar estas paginas cuando muera un Mr. Meeseek.

También seteamos su ss de nivel 0 para que tenga un selector de segmento de datos con RPL 0, mientras que los demas segmentos serán de datos o código según corresponda y con RPL 3 en todos los casos. Seteamos también tanto el ebp como el esp de esta tss para que apunten a la base de la segunda página mapeada. Activamos las interrupciones poniendo el valor del atributo eflags en 0x202 y por último activamos el bit de presente asociado a esta tss.

3. Implementar la lógica de la rutina de interrupciones move (123) para mover a un Meeseeks. Verificar que el remapeo y asimilación de semillas se realice de forma correcta. Comenzar implementando sin restricciones de movimiento.

   Para este punto, checkeamos en la rutina de interrupciones que la syscall no sea llamado por Rick o Morty. De ser asi, se da por ganador a el jugador contrario y se termina el juego de manera inmediata.

   En el caso que haya sido llamado por un Mr Meeseek, se llamara a la función `mover_meeseek` con las coordenadas pasadas por parametro.

   Esta función calculará la dirección tanto en x como en y nuevas, teniendo en cuenta que el wrap around esta permitido. Se checkeará si en estas nuevas coordenadas tenemos una mega semilla para asimilar. De ser asi, despues de asimilarla, se mata al meeseek en cuestión. De no haber una semilla, se pasará a mover el código de Mr. Meeseek a la nueva posición del mapa usando la función `mover_codigo_meeseek`.

   Esta función calculara la nueva dirección física en la que se debe copiar el meeseek, y mapeara el código viejo a esta nueva dirección. Para hacer esto mapeamos de manera temporal en el cr3 actual, la dirección física previamente mencionada usando identity mapping para copiar el código del meeseek, luego desmapeamos la dirección y luego remapeamos la nueva dirección física a la virtual que teníamos antes, concluyendo así la lógica para mover un meeseek.

4. Implementar la lógica de la rutina `look` (100) para buscar `Mega Semillas`.

   Esta interrupción depende de dos funciones en `sched.c`: `semilla_x` y `semilla_y`.

   En ambas funciones se verifica si la tarea que utilizo este servicio fue Rick o Morty, si fuera ese el caso entonces devolveria -1. Luego se recorren todas las semillas no consumidas del mapa y se calcula la distancia manhattan con respecto al meeseek invocador de este servicio, devolviendo la la coordenada x o y (depende de la función) de la semilla de menor distancia al meeseek. En el caso de haber mas de una semilla a la misma distancia se devuelven las coordenadas de la primera con respecto al orden de creación de las semillas.

   Dado que nuestro mapa es de 40 * 80 cuadrados y los meeseeks no realizan movimiento diagonal. Si queremos saber la distancia entre dos objetos dentro del mapa basta con tomar la diferencia absoluta entre sus coordenadas horizontales (abs(meeseek.x - semilla.x)) y sumarla a la diferencia absoluta entre sus coordenadas verticales (abs(meeseek.y - semilla.y)). A esta distancia se la conoce como distancia manhattan.

5. Implementar la lógica de la rutina `user_portal_gun` (89) para usar el arma de portales.

   En la rutina, verificamos que la syscall no sea llamada por un jugador. De ser así, se da por ganador al jugador contrario y se termina el juego de forma inmediata.

También verificamos que el meesseek no use más de una vez el portal gun, debido a la restricción del enunciado.

Finalmente, si es su primera vez usando el portal gun, usaremos la función `rand()`, para generar valores aleatorios y usaremos la funcion move_meeseek para moverlos con esas coordenadas.

6. Agregar restricciones de movimiento a la syscall mover.

   Agregamos a los meeseeks un atributo que da la distancia máxima que pueden recorrer los meeseeks. Empezando todos con 7 cuando se crean. Y tenemos un array de ticks de reloj que contabilizan cuantos ticks llevan. Cuando se crea un meeseeks el tick de reloj correspondiente a este se inicializa con 2. Y cada vez que se acaba su tick de reloj, bajamos su distancia máxima en 1, hasta un mínimo de 1, luego restauramos sus ticks de reloj.

   En la syscall move, checkeamos antes de proseguir con el código que la distancia que debemos movernos no exceda la distancia máxima.

7. Implementar la lógica de finalización del juego.

   Para terminar el juego, tenemos las verificaciones en las excepciones y en las syscalls. Para una finalizacion que no tengan que ver con lo mencionado anteriormente, tendremos un sistema de puntajes.

   Como dijimos anteriormente, cuando se crea un Mr Meeseek o un Mr Meeseek se mueve y estan parados en una Mega semilla, estas son asimiladas (no creando el meeseek o matandolo segun corresponda) y se da el puntaje al jugador dueño del Meeseek. Ahora, ademas de eso, verificaremos que todas las semillas hayan sido encontradas. De ser asi, el juego finaliza sentenciando al ganador segun el puntaje de cada uno. Si ambos tienen el mismo puntaje, se considera un empate. De cualquiera de las 3 formas (Gana Rick, Gana Morty, Empate) el juego termina.